MASTER EN SCIENCES INFORMATIQUES

Outils de manipulation d'expressions algébriques pour un didacticiel de la dérivation

Spelmans, Yves

Award date:
1991

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

OUTILS DE MANIPULATION D'EXPRESSIONS ALGEBRIQUES POUR UN DIDACTICIEL DE LA DERIVATION

par Yves SPELMANS

Promoteur:
Professeur Cl. Cherton

Mémoire présenté en vue
de l'obtention du titre
de Licencié et Maître
en Informatique

Année académique 1990-1991

A la mémoire de notre regretté Papa qui suivait de près la réalisation de ce mémoire mais qui vient de nous quitter prématurément.

Nous tenons à remercier le Professeur Cl. Cherton pour ses conseils et sa disponibilité.

Nous adressons notre reconnaissance à Monsieur Albrecht, notre employeur, pour sa grande compréhension et sans qui nous n'aurions eu le temps de mener à bien cette étude.

Nous adressons également toute notre gratitude à Monsieur et Madame Pilette ainsi qu'à tous mes proches qui m'ont beaucoup soutenu, dans ces moments difficiles et sans qui, la réalisation de ce travail n'aurait été possible.

TABLE DES MATIERES

INTRODUCTION

I. OBJECTIF

L'objectif de la présente étude est l'implémentation sur ordinateur d'un didactitiel pour l'enseignement de la dérivation des fonctions.

II. MOTIVATIONS

Le choix du présent travail répond à plusieurs raisons :

- premièrement, confronter les bases que nous avons acquises durant nos études avec un cas concret d'analyse d'un problème peu traité dans la littérature, en l'occurrence l'enseignement de la manipulation d'objets mathématiques complexes.
- ensuite, le désir de réaliser un projet qui puisse être source de créativité et autant que possible, donner des résultats encourageants.
- enfin, l'intérêt que nous avons tous à améliorer la connaissance dans l'approche d'un sujet et l'espoir de contribuer à la réalisation d'un projet qui trouvera un prolongement.

III. NATURE ET IMPORTANCE DU SUJET

Lorsque nous avons perçu pour la première fois le sujet, nous étions loin de nous douter de l'étendue et de la complexité des facettes qu'il revêtait.

L'enseignement n'est déjà pas une tâche triviale en soi. Il est la rencontre de plusieurs éléments qui une fois mis ensemble peuvent s'influencer l'un l'autre et atteindre une extrême complexité. Ce sont:

- l'enseignant;
- l'élève(s)

- la matière à enseigner
- les objectifs à atteindre et le contexte.

L'interpénétration de ces différents acteurs fait intervenir tant d'aspects de la psychologie humaine qu'il n'est pas facile d'en déceler tous les mécanismes. L'élève réagira en fonction de la personnalité du professeur, de la matière à acquérir et des objectifs qu'il s'est ou qu'on lui a assignés. Le professeur adaptera sa pédagogie en fonction de la complexité de la matière et de son auditoire. La matière également peut revêtir des aspects différents en fonction des autres partenaires de la relation...

Il n'existe d'ailleurs pas une, mais bien mille et une façons d'enseigner. Une méthode peut s'avérer bonne et efficace dans un cas et désastreuse dans un autre.
Dans le choix d'une méthode, chacun des acteurs joue un rôle important.

Que dire alors de sa transposition sur machine? L'aspect déterministe et purement mécanique de l'ordinateur n'est-il pas en-soi une porte qui ferme l'accès de l'enseignement à l'informatique ou, au contraire, peut-il devenir source de réflexions et ouvrir de nouvelles voies à la méthodologie de l'éducation?

Dans le cadre d'un enseignement assisté par ordinateur (E.A.O.), l'espace des contraintes est sans doute moins vaste que dans une relation professeur-élève. L'environnement que constituent le clavier et le petit écran conditionne fortement la réceptivité de l'élève qui sait presque instinctivement quels sont les limites de la relation qu'il peut avoir avec la machine. Cependant, cette relation existe bien et devient parfois si étroite qu'il arrive de voir des utilisateurs parler (invectiver) à leur petit écran.
Les contraintes sont sans doute également moins élevées du fait qu'il n'y a plus qu'une personne en présence.

De plus, l'utilisateur peut se fixé et revoir son rythme ainsi que ses objectifs d'apprentissage.
D'un autre coté, la réaction d'un élève peut être tellement imprévisible qu'il y aura toujours une limite à l'adéquation de la réponse de la machine quelque que soit le degré de sophistication atteint par le logiciel.

A coté de l'aspect "pédagogie à inculquer à la machine", le projet de départ touche un élément incontournable, qui est la matière à enseigner. Il ne s'agit pas ici, de réécrire un cours sur la dérivation des fonctions, ce que font pour d'autres matières, et avec brio, bon nombre de didacticiels conventionnels, mais bien de les maîtriser pour rendre le dialogue machine-élève plus riche. Cette maîtrise de la dérivation passe par le contrôle d'autres concepts tels que la représentation et la manipulation des expressions mathématiques.

Face à un projet prenant une telle ampleur et refusant de mener "à mal" une réalisation insignifiante qui tomberait dans l'oubli, nous nous sommes proposés de réaliser une première approche du problème qui pourrait trouver un prolongement dans des réalisations ultérieures. C'est pourquoi nous exposerons ci-après la façon dont nous avons limité notre travail ainsi que la méthodologie que nous nous sommes efforcés de suivre.

IV. METHODOLOGIE

Rappelons tout de suite que ce mémoire se rapproche de la catégorie des "Mémoires projets" en ce sens que l'étude que nous faisons du sujet est destinée à être poursuivie, voire, nous l'espérons, étendue à d'autres finalités.
Notre tâche sera donc avant tout axée sur une approche concrète de la viabilité du projet final, plus que sur son aboutissement immédiat.

Afin de clarifier ce que nous venons d'écrire, voici, en bref, la manière dont nous avons effectué notre travail:

- nous allons très brièvement donner l'image que nous nous faisons du sujet dans son ensemble pour préciser le cadre dans lequel nous avons étudié notre projet et de justifier le choix des réalisations que nous avons menées à bien; nous verrons que ces réalisations ne concernent pas directement le thème de l'E.A.O. mais plutôt ceux de l'algèbre et de l'analyse mathématique.
- nous détaillerons ensuite les choix et les stratégies impliqués dans les outils que nous avons développés;
- nous nous efforcerons enfin de donner les recommandations nécessaires à l'insertion et l'utilisation de nos outils dans une étude qui constituerait la suite ou une application de ce présent mémoire.
  Ceux-ci ont été développés en Turbo Pascal sur P.C. qui est, à notre sens, le support le plus adéquat pour cette classe d'application.

Notre travail ne repose sur aucune expérience antérieure et ne fera pas sans cesse référence à une bibliographie sur laquelle il ne s'est pas appuyé. Il représente d'ailleurs plus une réalisation qu'une étude de faisabilité.

V. DIFFICULTES RENCONTREES

Tout d'abord, nous avons éprouvé beaucoup de difficultés dans la recherche d'informations sur le thème de la représentation et de la manipulation des expressions algébriques et, si le problème a déjà été traité dans des implémentations plus importantes de simulateurs mathématiques, nous n'avons pas eu à notre disposition les outils de bases pour manipuler de pareilles expressions, or, ces outils constituant un élément indispensable à la poursuite de l'étude, il a fallu d'emblée se les donner.

Ceci donnant encore plus d'ampleur au projet initial, nous avons volontairement limité notre étude et nous sommes fixé comme objectif, la constitution d'outils essentiels à la poursuite du projet.

Une autre difficulté a été la maîtrise des problèmes par approches successives; c'est à dire, que sont apparues en cours de réalisation, des difficultés non soulevées lors de l'analyse initiale et il a donc fallu plusieurs fois réadapter la stratégie des algorithmes. Dans cet ordre d'idée, il est évident que si l'implémentation était à refaire, nous changerions certains choix et gagnerions en efficacité.

CADRE DE L'ETUDE

I. A PROPOS DES DIDACTICIELS

Le principe de fonctionnement de beaucoup de didacticiels répond à deux phases:

a) *la première phase* consiste en l'instruction; elle donne à l'utilisateur la possibilité de lire, non dans un livre, mais à l'écran, un exposé sur la matière qu'il doit apprendre. Cette phase peut être plus ou moins sophistiquée et permettre au lecteur de passer d'un point à l'autre de la matière sans nécessairement suivre un ordre chronologique qui lui serait imposé. Cette phase consiste en fait en un parcours dans un réseau sémantique.

b) *la deuxième phase* procède à une série de tests sur l'acquisition de la matière enseignée. L'utilisateur doit répondre à des questions de manière non ambiguë de telle sorte que le programme puisse interpréter ses réponse et en vérifier l'exactitude.
Pour lever l'inhérente ambiguïté présente dans la réponse de l'élève plusieurs stratégies sont possibles:
   - une série de réponses potentielles sont parfois proposées à l'élève, et celui-ci n'a plus qu'à choisir celle qui lui semble exacte.
   - une seconde possibilité serait de doter le logiciel, d'un processus d'analyse de la réponse.

Nous tenons pas à insister plus loin sur la typologie des didacticiels ni même en faire une ébauche plus poussée. Elle ne nous aidera pas dans la suite de notre étude et pourrait d'ailleurs représenter à elle seule une étude à

part entière. Cette étude serait d'ailleurs bien mieux menée que nous, par un féru de pédagogie de l'enseignement.

II. VERS DE NOUVEAUX OBJECTIFS

Au tout premier jours de notre étude, nous avons imaginé de commencer notre travail par la réalisation de la première phase. Nous y serions parvenu, avec plus ou moins de réussite, à l'aide de logiciels comme "HyperCard", ou de bibliothèques de procédures déjà très poussées comme "Turbo Professional", ainsi que de nos bonnes notions de l'analyse mathématique. Nous aurions pu ainsi expliquer à l'élève, ce qu'est une dérivation.
Nous avons vite estimé que cela manquerait d'intérêt et nous avons postposer la réalisation de cette phase.

Nous avons considéré (et nous ne sommes pas le seul) que l'étude et le test de la connaissance de la dérivation, passe par la résolution d'exercices, nous pensions alors nous constituer une bibliothèque d'exercices sur la dérivation à laquelle nous adjoindrions les différentes étapes de résolution qui seraient associées à des pointeurs vers les différentes règles utilisées. Ces règles pourrait à leur tour pointer, à la manière des réseaux sémantiques, vers un exposé sur le thème en question.
Nous aurions pu, de cette manière, montrer à l'élève, comment dériver une fonction.

Dans ce processus d'apprentissage, l'élève reste néanmoins passif et pas un professeur d' analyse mathématique, n'oserait affirmer que regarder quelqu'un résoudre des exercices constitue une pédagogie suffisante.
Cette façon de procéder serait intéressante pour la première phase d'action de notre didactitiel mais, nous la postposons également car elle ne constitue pas une difficulté en soi.

Nous nous sommes alors tout naturellement tournés, vers la deuxième phase qui consiste en la vérification de la connaissance de la dérivation. Ici il n'est plus question de préparer une série de questionnaires (à choix multiples par exemple); le type de matière à enseigner ne s'y prête pas très bien.

Nous préférerons de loin, adjoindre au didacticiel, un processus d'analyse des solutions proposées par l'élève. Ce processus est extrêmement complexe car la manipulation et la représentation des expressions algébriques est ambiguë. Au départ d'une expression ont peu lui trouver une infinité d'expressions équivalentes. Les étapes d'une dérivation constitue également un processus ambigu. Chaqu'un, selon son intuition, procédera différemment.

Nous essaierons, dans la suite de notre travail, de construire les mécanismes qui permettront d'analyser et de valider les solutions proposées par l'élève. Il faudra d'une part que notre programme puisse effectuer une dérivation et qu'il puisse, par ailleurs comparer sa solution avec celle fournie par l'élève. Nous pourrions alors dire à l'élève qu'il connaît la dérivation.

CONCEPTS ET STRATEGIE MIS EN JEU

I. INTRODUCTION

Dans ce chapitre, nous allons essayer de développer une stratégie de représentation, de simplification et de dérivation des expressions algébriques car, de toute évidence, la représentation d'une expression par la juxtaposition de caractères et l'imitation des mécanismes de raisonnement humain, sont inadéquats pour une modélisation informatique.

La représentation généralement adoptée, pour ce type de modélisation, est une représentation sous forme d'arbres. Un arbre a, en effet, la propriété intéressante d'indiquer les relations existant entre les différents éléments qui le constituent, sans qu'il soit nécessaire de le considérer dans son ensemble. De plus, les arbres se prêtent bien à l'application de processus systématiques.

Nous allons ,dans le présent chapitre définir, des arbres et des outils pour les manipuler, qui puissent répondre à nos besoins, notre objectif final étant, outre la dérivation, de pouvoir comparer des expressions et d'obtenir la probabilité, la plus élevée possible, de reconnaître quand elles sont équivalentes.

II. L'ANALYSE SYNTAXIQUE

1) INTRODUCTION

Avant la transposition de l'expression en arbre algébrique, ce que fait le traducteur d'expression (voir p.27), il y a toujours une précondition très sévère: l'expression mathématique doit respecter une certaine structure syntaxique. Elle doit appartenir à l'ensemble d'expressions que l'on s'est défini et doit donc en avoir les caractéristiques et en vérifier les propriétés. La traduction n'est possible que si l'on a un contrôle de la syntaxe de l'expression. L'analyseur syntaxique joue ce rôle; il s'assure de l'appartenance de l'expression à notre système.

En amont de l'analyseur syntaxique, il y a *l'analyseur lexicographique*. C'est le premier passage de la *représentation externe* de nos expressions vers leur *représentation interne*. Celui-ci découpe séquentiellement le ruban de caractères initial, en petites chaînes qui peuvent donner naissance aux éléments atomiques de notre grammaire: *les symboles terminaux*. Il repère également les chaînes de caractères incompatibles avec notre système de représentation.

La *grammaire* de notre espace d'expression est définie par un ensemble de *classes syntaxiques*, chacune d'entre elles étant définie comme, une concaténation d'éléments d'autres classes syntaxiques et de symboles terminaux qui sont les constituants atomiques de la représentation interne de nos expressions. Les relations entre les différentes classes syntaxiques sont définies au moyen de *règles de productions*. Celles-ci sont des transformations sur l'ensemble des classes syntaxiques et permettent de passer d'une *classe syntaxique* à une autre. Elle définissent la syntaxe de notre ensemble.

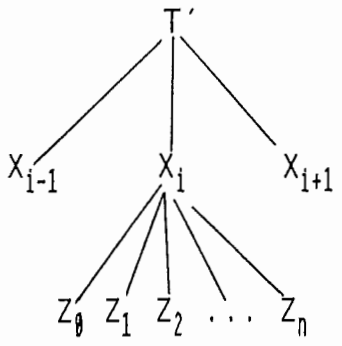
L'analyseur syntaxique reçoit de l'analyseur lexicographique une suite de symboles terminaux, image de notre expression de départ, et teste si leur enchaînement est syntaxiquement correct. Pour cela il essaie d'obtenir, en leurs appliquant des règles de production, une classe syntaxique particulière représentant une expression syntaxiquement correcte appelée *"la variable distinguée"*, . S'il l'obtient, on peut en déduire que notre expression est grammaticalement correcte.

2) L'ARBRE D'ANALYSE

Une façon intéressante de représenter une expression en mémoire serait une représentation arborescente, la notion d'arbre étant aisément implémentable sur ordinateur.
Construisons cet arbre que nous appellerons *"arbre d'analyse"*ou *"arbre syntaxique"*:
1) A la variable distinguée, on associe un noeud.
2) Si "T" est un arbre d'analyse, et s'il existe une feuille de "T" dont l'élément est "X_i" tel que correspond "X_i" à la partie gauche d'une règle de production:

$$X_i ::= Z_0\ Z_1\ Z_2\ \ldots\ Z_n$$

alors, on peut construire un nouvel arbre d'analyse "T'" à partir de "T" où la feuille X_i devient racine d'un sous arbre:



3) On applique de nouveau le point "2" aux autres feuilles de l'arbre ainsi qu'aux feuilles Z_i jusqu'à l'obtention de feuilles qui soient des symboles terminaux.

3) EQUIVALENCE EXPRESSION - ARBRE D'ANALYSE

On désire établir une correspondance biunivoque entre l'expression mathématique de départ et sa transposition sous forme d'arbre d'analyse.

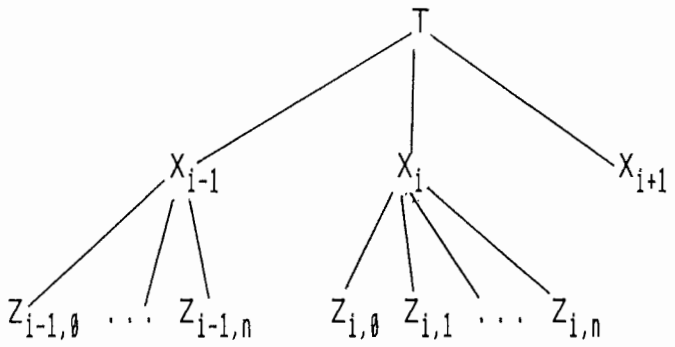
a) Si, on appelle: "*forme de phrase*" toute concaténation d'éléments (classes syntaxiques) du langage formant des textes syntaxiquement corrects, et "*phrase*" une forme de phrase composée uniquement de symboles terminaux alors, partant d'une phrase ou d'une forme de phrase, on peut remplacer une séquence d'éléments formant la partie droite d'une règle de production par l'élément formant la partie gauche de la même règle de production et obtenir une nouvelle forme de phrase relative à la même expression. On peut opérer de tels remplacements jusqu'à obtention de la forme de phrase particulière correspondant à la "*variable distinguée*".

b) Considérons la relation "*A la gauche de (gd)*", qui s'applique à deux feuilles d'un arbre d'analyse et qui est définie comme suit:

> *Si deux feuilles x et y sont directement descendantes d'un même noeud père, alors x gd y signifie que x précède y dans la règle de production.*

On remarque que si on considère toutes les feuilles de l'arbre, on peut les organiser de façon à obtenir un ordre par rapport à la relation "*A la gauche de (gd)*".

Soit un arbre:



On obtient aisément:

$Z_{i-1,0}$ gd ... gd $Z_{i-1,n}$ gd $Z_{i,0}$ gd $Z_{i,1}$ gd ... gd $Z_{i,n}$ gd X_{i+1}

Cette présentation des feuilles est appelée *"produit"* ou *"yield"* de l'arbre syntaxique.

On peut montrer que le *produit* d'un arbre d'analyse, à chaque étape de sa construction, est identique à la *forme de phrase* dont il est issu. On peut ainsi établir une correspondance entre l'arbre syntaxique et la forme de phrase originale.

Mais cette correspondance ne sera biunivoque que si à chaque *phrase* correspond un arbre qui est unique ce qui n'est assuré que si la grammaire est non ambiguë.
Malheureusement, la propriété de non ambiguïté d'une grammaire est une question indécidable; c'est-à-dire qu'il n'existe pas de programme pouvant répondre, pour toutes grammaires, à la question: "Cette grammaire est-elle ambiguë", mais il est néanmoins possible de construire des tests, applicables à notre syntaxe, qui permettent d'affirmer qu'une grammaire est non ambiguë.

4) ANALYSEUR DE TYPE BOTTOM-UP

En fait, nous procéderons différemment pour construire notre arbre, mais les concepts développés plus haut restent valides.

Au lieu de partir de la variable distinguée, puisque c'est justement ce que l'on veut obtenir, et de développer tous les noeuds jusqu'à l'obtention des symboles terminaux, nous édifierons l'arbre d'analyse à l'envers. On a une *phrase* formée de *symboles terminaux*, à partir de laquelle on va essayer de former un arbre:

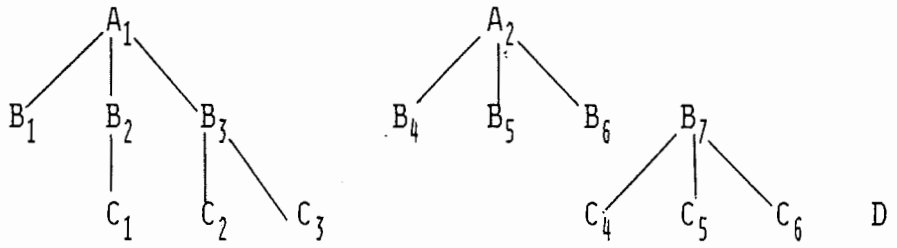
a) Chaque symbole terminal sera représenté par un noeud et l'ensemble de ces noeuds constituant une forêt d'arbres sera ordonné par la relation *"a la gauche de"*..

b) A une séquence de noeuds:" X_1 gd X_2 gd X_3 gd ...gd X_n" concordant avec la partie droite d'une règle de production, correspond une classe syntaxique "Y" concordant avec la partie gauche de la même règle de production.
L'application de cette règle de production, pour passer d'une forme de phrase à une autre, sera matérialisée par la création d'un nouveau noeud "T" et par des arcs pointant de ce nouveau noeud vers les racines "X_i" des arbres ayant servis à cette construction.

c) En réappliquant le point "b", les différents arbres de départ sont donc peu à peu agglomérés en arbres de plus en plus larges et nous obtiendrons la variable distinguée lorsque la forêt d'arbres se sera réduite à un seul arbre. Si à un moment de la construction de cet arbre, il n'est plus possible d'appliquer une règle de production, et que nous sommes encore en présence d'une forêt d'arbres, c'est que notre expression de départ ne respecte pas notre grammaire.

La notion de *produit* doit également être transposée à l'envers: Il s'agit maintenant de la représentation ordonnée, non plus des feuilles, mais bien des racines des différents arbres constituant notre forêt.

ex:



donnera comme produit: A_1 gd A_2 gd B_7 gd D

Des critères de non ambiguïté nous imposent de ne pas avoir le choix entre l'application de plusieurs règles de production lors de la construction des nouveaux noeuds, ce qui pourrait nous conduire au départ d'une seule expression à la possibilité de construction d'arbres multiples, syntaxiquement identiques mais, différents.
Ceci nous amène à convenir de certaines règles, dans la définition de notre grammaire ainsi que dans le choix des règles de production lors des différentes étapes de la construction de l'arbre. En fait, pour lever l'ambiguïté, il faut même supprimer cette possibilité de choix.

- *Deux règles de production ne peuvent jamais avoir la même partie droite, afin que la construction du noeud père soit univoque.*

- *Si la partie droite d'une règle de production constitue un suffixe d'une autre règle de production, plus longue, on ne peut utiliser la règle la plus courte que s'il n'y a pas moyen d'utiliser la plus longue.*

- *Nous construirons nos nouveaux noeuds toujours le plus à droite possible; c'est-à-dire que le noeud qui représente l'extrémité droite du produit de notre forêt interviendra prioritairement dans l'application d'une règle de production.*

ex:

soit: X_1 gd X_2 gd X_3 gd ... gd X_i gd ... gd X_{n-1} gd X_n le produit d'un arbre à un stade quelconque de sa construction et:

Y_1 ::= X_2 X_3

Y_2 ::= X_3 ... X_i

Y_i ::= X_i ... X_{n-1}

les règles de production applicables au *yield* de notre forêt d'arbre, nous nous imposerons d'appliquer la règle qui fait intervenir les éléments les plus à droite possible: Y_i ::= X_i ... X_{n-1}, dans notre exemple.

En partant du noeud le plus à droite "D", on considère ses prédécesseurs et, si l'on trouve une séquence de noeuds:" X_1 gd X_2 gd X_3 gd ... D ", correspondant à la partie droite d'une règle de production, on l'applique, sinon, on tente de former une telle séquence en prenant l'élément à gauche de D, comme extrémité droite.

III. IMPLEMENTATION DE L'ANALYSEUR SYNTAXIQUE

1) LA GRAMMAIRE

Nous avons établi notre grammaire sur base des expressions que nous voulions pouvoir représenter, en tenant compte à priori du critère de non-ambiguïté énoncé plus haut et avons testé cette non-ambiguïté à l'aide d'autres tests appliqués à posteriori, lors de la construction de la table "S.R.E." (voir p.22).

A) les symboles terminaux

Ce sont les classes syntaxiques atomiques de notre grammaire, fournies telles quelles à l'analyseur, par l'analyseur lexicographique:

- <RAT>,<VARIABLE>,<ID_FONC>;
- <CPLUS>,<CMOINS>,<CFOIS>,<CDIV>,<CEXP>;
- <GPAR> et <DPAR>

B) Règles de grammaire ou de production (en BNF)

Ces règles définissent de nouvelles classes syntaxiques par la concaténation d'éléments d'autres classes syntaxiques et de symboles terminaux:

```
<EXPRESS> ::=<EXPRESS><CPLUS><TERME>
          |<EXPRESS><CMOINS><TERME>
          |<TERME>

<TERME>   ::=<TERME><CFOIS><FACTEUR>
          |<TERME><CDIV><FACTEUR>
          |<FACTEUR>

<FACTEUR> ::=<FACTEUR><CEXP><CVALEUR>
          |<CVALEUR>

<CVALEUR> ::=<GPAR><EXPRESS><DPAR>
```

|<CMOINS><GPAR><EXPRESS><DPAR>
|<FONC>
|<CMOINS><FONC>

|<RAT>
|<CMOINS><RAT>
|<VARIABLE>
|<CMOINS><VARIABLE>

<FONC> ::=<ID_FONC><CVALEUR>

L'application d'une de ces règles permet de remplacer l'élément de gauche par la séquence des éléments de droite (ou le contraire).

2) L'ALGORITHME

A) Introduction

En fait, il n'est pas indispensable, lors du choix d'une règle de production, de considérer le noeud le plus à droite possible mais, il faut être certain, en partant d'un noeud quelconque "T_i", que ce noeud ne puisse appartenir, avec celui (T_{i+1}) ou ceux ($T_i \ldots T_{i+n}$) qui le suivent à droite, à la partie droite d'une règle de production. Pratiquement, on s'assurera de la non existence de séquence "$T_i\ T_{i+1}$" parmi les parties droites des règles de production.

Partant de ce principe, on peut se positionner sur l'élément le plus à gauche, qui est également le premier fourni par l'analyseur lexicographique, et avancer vers la droite jusqu'à ce que la condition soit respectée. On ne devra pas, comme cela, considérer l'ensemble de l'expression avant de commencer à construire l'arbre. A chaque étape de la construction de l'arbre, la condition devra être vérifiée et il

sera parfois nécessaire de se repositionner sur un noeud plus à droite avant d'appliquer une nouvelle règle de production.

Si la grammaire est non ambiguë, il ne sera pas non plus nécessaire de se repositionner sur un noeud plus à gauche; c'est-à-dire de revenir en arrière.
Nous nous trouverons alors dans le cas d'un *analyseur à précédence faible*.

B) Structure des données

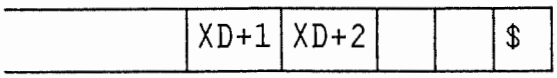
a) Le *stack* (pile)

Cette structure est capable de chaîner les noeuds formant le yield de notre forêt d'arbres et traduit facilement l'ordre induit par la relation "gd". L'élément "X_d" le plus à droite de la partie du produit à laquelle on va essayer d'appliquer une règle de production se trouve au-dessus de la pile, celui immédiatement à gauche venant juste en-dessous et ainsi de suite jusqu'au premier élément du produit.

$	X1	X2	X3			XD	

b) L'*input* (2ème pile)

Cette structure, semblable au *stack* contient tous les noeuds formés des symboles terminaux, à droite de "X_d" dans le yield; c'est-à-dire: les noeuds qui ne sont pas encore pris en considération pour l'application d'une règle de production et qui sont donc nécessairement des symboles terminaux.
Ici, contrairement au stack, les éléments les plus à gauche forment le sommet de la pile.



c) on notera ici que la lecture du stack depuis l'élément le plus ancien jusqu'au plus récent, suivie de la lecture de gauche à droite de l'input donne le yield de notre forêt d'arbres.

C) Les étapes de l'algorithme d'analyse

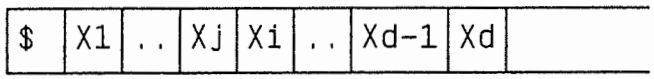
a) Invariant:

Après chaque étape de la construction de l'arbre, si le texte de départ est syntaxiquement correct, la lecture du stack depuis l'élément le plus ancien jusqu'au plus récent, suivie de la lecture de gauche à droite de l'input constitue une forme de phrase de l'expression de départ.

b) 1ère étape: Action de *REDUCTION*:

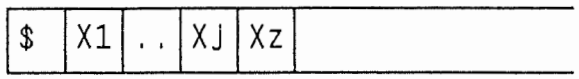
On applique une règle de production aux éléments se trouvant au dessus de la pile:

Xz ::= Xi ... Xd-1 Xd



et on obtient un nouvel état de la pile:
les noeuds "Xi ... Xd" formant le dessus de la pile et correspondant à la partie droite de la règle de production, sont remplacés par un nouveau noeud "Xz" qui correspond à la partie gauche de cette même règle et qui prend le sommet de la pile. La REDUCTION a pour effet de transformer et de diminuer le sommet de la

pile. Les éléments inférieurs de la pile reste inchangés.



c) 2ème étape: Action *SHIFT*:

Cette action consiste à déplacer l'élément formant le sommet de l'input, au sommet du stack.

d) Le choix de l'action:

La question est de savoir, quand il faut effectuer une réduction, ou un shift. Pour répondre à cette question, nous verrons que seuls les éléments constituant le sommet des deux piles interviennent. Il suffit dès lors de se donner une table de décision à deux entrées pour connaître immédiatement en fonction de ces deux éléments, l'action à accomplir ou, s'il y a erreur.
Trois situations sont possibles:

1) Le sommet du stack et de l'input ne figurent ensemble et dans cet ordre, dans aucune partie droite de règle de production; on peut donc appliquer une règle de production aux éléments supérieurs du stack: ==> action REDUCTION.
   Cette réduction n'est possible que si l'on peut trouver une séquence parmi les éléments supérieurs de la pile identique à la partie droite d'une règle de production, sinon:
2) ERREUR. On ne parviendra jamais à réduire notre forêt d'arbres à un arbre unique et l'expression de départ n'est pas syntaxiquement correcte.
3) Les sommets des deux piles apparaissent dans le même ordre dans la partie droite d'une règle de production: ==> action SHIFT.

e) remarque à propos de l'Input

Comme, seul le sommet de cette $2_{ème}$ pile intervient dans l'algorithme proprement dit, il n'est pas nécessaire que toute l'expression à analyser soit, dès le départ, présente sous forme de symboles terminaux dans l'input. Autrement dit, l'input peut recevoir, de l'analyseur lexicographique, lors de chaque action shift, l'élément formant son nouveau sommet.

3) CONSTRUCTION DE LA TABLE S.R.E. ET TEST DE LA GRAMMAIRE

De toutes les considérations précédentes, nous établirons quelques règles qui, appliquées systématiquement, nous permettent de créer la table de décision.
Rappelons d'abord quelques principes:

a) *Deux règles de production ne peuvent jamais avoir la même partie droite.*

b) *Si la partie droite d'une règle de production constitue un suffixe d'une autre règle de production plus longue, on ne peut utiliser la règle la plus courte que s'il n'y a pas moyen d'utiliser la plus longue.*

c) *Si le sommet du stack et de l'input peuvent apparaître ensemble dans la partie droite d'une règle de production, il y a SHIFT.*

d) *Lorsque le stack est vide, il y a forcément un SHIFT.*

e) *Lorsque l'input est vide, il ne peut plus y avoir que des REDUCTIONS.*

Soit "X", le sommet du stack et "Y", le sommet de l'input, le couple (X,Y) de la table sera noté "R" si X et Y donnent lieu à une action REDUCTION et "S" s'ils donnent lieu à une

action SHIFT et E dans les autres cas qui donnent lieu à une ERREUR. Des principes énoncés ci-dessus et de considérations qui sortent du cadre de la présente étude, on peut déduire de notre grammaire (voir p.17), les règles de construction de la table, ci-dessous:

1) <VIDE> $\underline{S}$ <EXPRESS>

2) Si X $\underline{S}$ Y et si il existe: $Y ::= Z_1\ Z_2\ \ldots\ Z_n$
Alors: $X\ \underline{S}\ Z_1$

3) Si il existe: $Y ::= Z_1\ Z_2\ Z_3\ \ldots\ Z_{n-1}\ Z_n$
Alors: $Z_1\ \underline{S}\ Z_2$
$Z_2\ \underline{S}\ Z_3$
. . .
$Z_{n-1}\ \underline{S}\ Z_n$

4) <EXPRESS> $\underline{R}$ <VIDE>

5) Si X $\underline{S}$ Y et si il existe: $X ::= Z_1\ Z_2\ \ldots\ Z_n$
Alors: $Z_n\ \underline{R}\ Y$

6) Si X $\underline{R}$ Y et si il existe: $Y ::= Z_1\ Z_2\ \ldots\ Z_n$
Alors: $X\ \underline{R}\ Z_1$

7) Si X $\underline{R}$ Y et si il existe: $X ::= Z_1\ Z_2\ \ldots\ Z_n$
Alors: $Z_n\ \underline{R}\ Y$

A l'aide de ces règles et de notre grammaire, nous pouvons construire la table S.R.E.
Notons que tout couple (X,Y) doit avoir une image unique dans la table, car une telle situation engendrerait des ambiguïtés ou nécessiterait des rétroparcours. Cette condition constitue d'ailleurs avec les principes énoncés plus haut une condition suffisante de non ambiguïté de la grammaire.

La table, une fois construite, ne constitue pas seulement un outil de base de l'algorithme, mais également un test de non-ambiguïté de notre grammaire qui, rappelons-le est une question indécidable. Nous avons nous-même procédé à plusieurs essais avant d'obtenir notre grammaire définitive (voir p.17).

Table SRE

	EXPRESS	TERME	FACTEUR	CVALEUR	FONC	RAT	VARIABLE	ID FONC	CPLUS	CMOINS	CFOIS	CDIV	CEXP	GPAR	DPAR	VIDE
<EXPRESS>	E	E	E	E	E	E	E	E	S	S	E	E	E	E	S	R
<TERME>	E	E	E	E	E	E	E	E	R	R	S	S	E	E	R	R
<FACTEUR>	E	E	E	E	E	E	E	E	R	R	R	R	S	E	R	R
<CVALEUR>	E	E	E	E	E	E	E	E	R	R	R	R	R	E	R	R
<FONC>	E	E	E	E	E	E	E	E	R	R	R	R	R	E	R	R
<RAT>	E	E	E	E	E	E	1	1	R	R	R	R	R	E	R	R
<VARIABLE>	E	E	E	E	E	E	1	1	R	R	R	R	R	E	R	R
<ID_FONC>	E	E	E	S	S	S	S	S	E	S	E	E	E	S	E	E
<CPLUS>	E	S	S	S	S	S	S	S	E	S	E	E	E	S	E	E
<CMOINS>	E	S	S	S	S	S	S	S	E	S	E	E	E	S	E	E
<CFOIS>	E	E	S	S	S	S	S	S	E	S	E	E	E	S	E	E
<CDIV>	E	E	S	S	S	S	S	S	E	S	E	E	E	S	E	E
<CEXP>	E	E	E	S	S	S	S	S	E	S	E	E	E	S	E	E
<GPAR>	S	S	S	S	S	S	S	S	E	S	E	E	E	S	E	E
<DPAR>	E	E	E	E	E	E	E	E	R	R	R	R	R	E	R	R
<VIDE>	S	S	S	S	S	S	S	S	E	S	E	E	E	S	E	E

Nous remarquerons que, puisque l'input est constitué uniquement de symboles terminaux, il n'est pas nécessaire de conserver dans le programme les colonnes de la table correspondant aux autres classes syntaxiques.

Les couples (x,y) de la table dont la valeur est "1" proviennent d'une tolérance, accordée à ces couples, à enfreindre à notre grammaire. Nous acceptons dès lors des expressions du genre "3 X" au lieu de "3 * x" ou,
"5 COS(X)" au lieu de "5 * COS(X)".
L'analyseur lexicographique insérera lui-même l'élément manquant "<CFOIS>" entre les deux éléments.

Nous aurions pu permettre un autre abus d'écriture pour les expressions du genre "X 2" au lieu de "X ^ 2", en donnant une valeur particulière au couple (<VARIABLE>,<RAT>) de la table.

Il serait également judicieux, surtout dans le cadre d'un didactitiel de pouvoir expliciter ce en quoi une expression est syntaxiquement non correcte. Il suffirait de remplacer les valeur "E" de la table S.R.E. par des valeurs particulières associées aux types des erreurs commises.
Nous pourrions alors retourner à l'utilisateur un message adéquat ainsi que la localisation de l'erreur, ce que nous ferons partiellement dans notre implémentation.

II. LE TRADUCTEUR D'EXPRESSION

1) INTRODUCTION

La finalité du programme n'est pas de valider la syntaxe d'une expression algébrique mais bien de se donner des outils de travail sur ces expression. Il s'agit ici de voir quels seront les mécanismes adoptés pour la transposition des expressions, de leur représentation externe, sous forme de chaînes de caractères, vers leur représentation interne, sous forme d'arbres algébriques.

Ce qui est remarquable, c'est que les étapes de l'algorithme de traduction sont identiques à celle de l'algorithme d'analyse si ce n'est, que lors d'une application d'une règle de production il n'y a pas un "remplacement d'éléments" mais bien une construction.
Dans le processus d'analyse, nous ne nous intéressions qu'au PRODUIT (voir p.13 et 14) de la forêt d'arbres, c'est-à-dire les racines de ces différents arbres. C'est d'ailleurs volontairement que nous utilisions les vocables "arbre" et "forêt d'arbres" qui englobent la notion de noeud isolés.
Dans le processus de traduction, ces notions reprennent toute leur dimension et à chaque application de règle de production sera adjointe une construction arborescente.

Lors de l'analyse syntaxique, nous avons largement discuté de la non ambiguïté de la grammaire. Nous voulions rejeter, par exemple, la possibilité d'obtenir, à partir d'une expression, des arbres multiples, même s'ils sont équivalents. Ces considérations n'étaient pas toutes indispensable, puisque le but de l'analyse était simplement de tester la syntaxe d'une expression, et c'est seulement ici qu'elles prennent toute leur importance.

Nous ne discuterons donc ici que des aménagements qu'il y a lieu d'apporter à notre analyseur syntaxique pour qu'il devienne, en plus, un traducteur d'expression.

2) SYNTAXE VALUEE

L'analyseur lexicographique ne se contentera plus de reconnaître les éléments de l'expression et de créer les symboles terminaux correspondants, il leur adjoindra en plus des attributs pour ne pas perdre les caractéristiques sémantiques de l'expression:

- aux éléments de type "<RAT>" qui correspondent aux nombres présents dans l'expression, on ajoutera un attribut fixant leur valeur.
- Aux éléments de type "<VARIABLE>", on accolera, deux nombres: un cofacteur ainsi qu'une puissance.
  Nous aurions pu également lui adjoindre un "nom" mais, nous avons, dans le cadre de la dérivation, choisi de ne traiter que les expressions à une seule variable.
- Aux éléments de type "<ID_FONC>" nous associons, comme pour les "<VARIABLES>" un cofacteur et une puissance. Nous leurs lions, en plus, un "nom de fonction" (cos,sin,tg,ln,...).

3) L'ALGORITHME

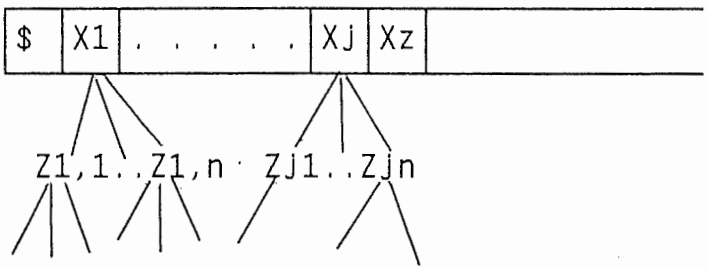
L'algorithme utilise les même structures de données que l'algorithme d'analyse, à savoir: le stack et l'input (voir p.19).
Les éléments qu'ils contiennent sont les racines des arbres en cours de construction et non plus des éléments isolés.

Il a recours aux mêmes étapes (voir p.20...) et exploite la même table de décision (la table S.R.E.).
L'application d'une règle de production lors d'une étape de réduction change cependant:

a) Il n'y a plus remplacement mais bien construction d'un nouvel élément, au dessus de la pile.

b) A chaque règle de production correspond une routine sémantique (les routines *REDUCE_i* dans notre programme)qui crée le nouvel élément correspondant à la partie gauche de la règle. Ce nouvel élément est la racine d'un arbre construit avec les éléments formant la partie droite de la règle.

c) Le nouveau sommet de la pile contient la racine de l'arbre nouvellement créé et cet arbre ne sera pas, en général, un noeud isolé, mais un arbre, et les fils de cet arbre ne seront pas des éléments du stack.



III. LA SIMPLIFICATION DES ARBRES

1) INTRODUCTION

La transposition des expressions, de leur représentation externe, sous forme de chaînes de caractères, vers leur représentation interne, sous forme d'arbres algébriques, ne constitue qu'un pas de notre modélisation. Lors de l'analyse syntaxique, nous avons largement discuter de la non ambiguïté de la grammaire. Nous voulions rejeter, par exemple, la possibilité d'obtenir, à partir d'une expression, des arbres multiples, même s'ils sont équivalents. Mais, si à partir d'une expression, on arrive à l'obtention d'un arbre unique, le processus reste cependant ambigu. L'ambiguïté provient de l'aspect sémantique des expressions: A chaque expression algébrique correspond une multitude de représentations algébriquement équivalentes:

a) Les opérateurs commutatifs "+" et "*" autorisent la permutation de leurs opérandes, ce qui entraîne, si l'expression contient un nombre "n" de tels opérateurs (n+1 opérandes), l'existence de "(n+1)!" expressions équivalentes donnant naissance à "(n+1)!" arbres algébriques sémantiquement équivalents mais néanmoins différents.
On pourrait, pour lever cette ambiguïté, se donner un ordre strict sur les opérandes de tels opérateurs.

b) Chaque élément algébrique est toujours décomposable d'une infinité de manières en s'exprimant en fonction d'expressions plus simples ou plus complexes; encore faut-il pouvoir juger la complexité des expressions
   - Le processus de factorisation constitue-t-il une simplification, dans quel cas, l'opération de distribution des opérateurs serait un processus de complexification? La réponse sera tantôt oui, tantôt non.

- La simplification des expressions répond à des mécanismes non monotones, tantôt d'éclatement, tantôt de regroupement, des éléments qui la constitue. Elle fait largement appel, dans sa mise en oeuvre, à l'intuition qui est un processus ambigu.

c) Certaines simplifications sont néanmoins évidentes, tel que le regroupement et la simplification d'éléments similaires, en fonction des opérateurs qui les séparent.

d) Il existe, pour chaque opérateur, des opérandes remarquables telles que les éléments neutres et les éléments absorbants. On peut facilement en tirer parti

e) D'autres, comme la simplification des expressions portant sur les fonctions circulaires, non pas été implémentées mais, on pourrait imaginer les exprimer en fonction de "Tg(x/2)" à l'entrée de notre algorithme de simplification, pour les reconvertir ensuite.

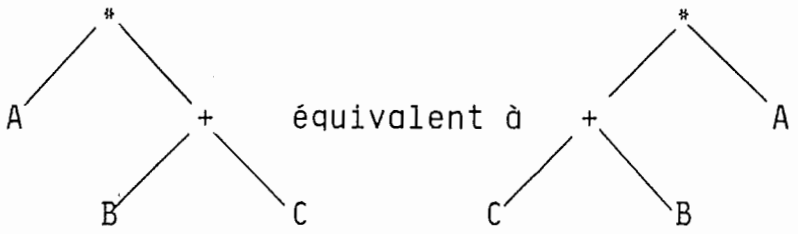
Notre objectif est donc ici, l'obtention d'un arbre unique à partir d'expressions multiples, mais sémantiquement identiques. Disons tout de suite que notre espoir est vain et que notre modélisation ne constituera qu'une approche du problème. Nous leverons la dimension du problème posée au point "a" et nous appliquerons les propriétés évoquées aux points "d" et "e".

2) NORMALISATION DES ARBRES

Si on considère un arbre algébrique constitués uniquement des opérateurs commutatifs "+" et "*", on vérifie aisément les deux propriétés de construction suivantes:

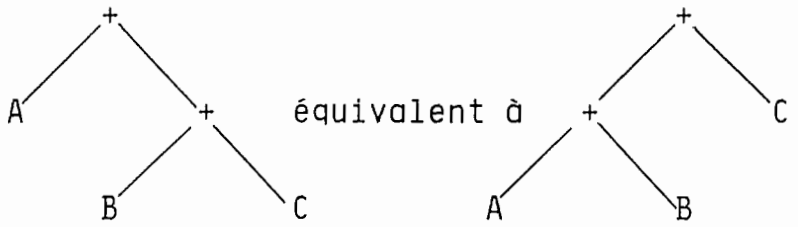
a) *Si l'on construit un nouvel arbre en intervertissant les branches gauche et droite de l'arbre de départ, on obtient un nouvel arbre algébriquement identique à l'arbre initial.*

ex:



b) *Si dans un arbre deux noeuds contigus sont constitués du même opérateur alors, si l'on construit un nouvel arbre en intervertissant des branches de ces deux noeuds de l'arbre de départ, on obtient un nouvel arbre algébriquement identique à l'arbre initial.*

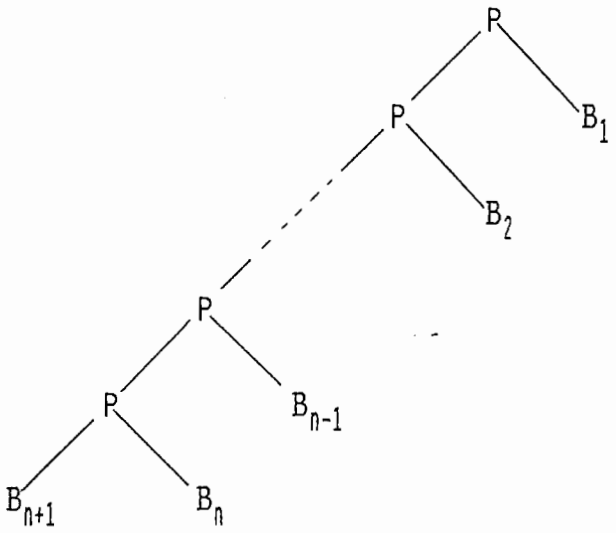
ex:



c) De la propriété "b" on peut déduire que *si dans un arbre, non plus deux, mais "n" noeuds contigus sont constitué du même opérateur, alors tout arbre construit par permutation des branches de ces "n" noeuds est algébriquement identique à l'arbre initial.* On remarque que le nombre d'arbres qu'il est possible de construire est "**(n+1)!**" ce qui correspond aux "**(n+1)!**" arbres sémantiquement identiques du point "a" de l'introduction.

c) Soit "T" un arbre quelconque ayant "n" noeuds contigus constitués du même opérateur "P". Construisons en

appliquant la règle "c" un arbre équivalant à "T", en prenant la convention suivante: *un noeud "P" ne peut avoir comme sous-arbre de droite un arbre dont la racine serait "P"*. Notre arbre sera nécessairement de la forme:



La suite $(B_1, B_2, \ldots, B_{n-1}, B_n)$ est une permutation des branche "B_i" des "n" noeuds considérés dans notre arbre de départ.

Si l'on pouvait se donner un ordre sur les "B_i", nous saurions comment représenter nos "**(n+1)!**" expressions sémantiquement équivalentes, de manière univoque et nous aurions résolu le problème soulever au point "a" mais uniquement pour les opérateurs "+" et "-".

3) SUPPRESSION DES OPERATEURS <CMOINS> ET <CDIV>

Les opérateurs non commutatifs "-" et "/" constitue dans nos arbres algébriques, des barrières que les éléments que les éléments situés à gauche ou à droite ne peuvent franchir. Il est indispensable, s'il l'on veut une représentation unique des expressions équivalente de dépasser cette difficulté.

a) Nous pourrions remplacer l'opérateur "-" par "+" si nous construisions avec la branche de droite, un arbre qui serait son opposé par rapport à l'élément neutre "Ø". "Trouver l'opposé d'un arbre" n'est pas une difficulté en soi, et peu être défini récursivement:

- Les feuilles pendantes d'un arbre sont nécessairement un "<RAT>" ou une "<VARIABLE>" dont l'opposé s'obtient en multipliant par "-1" l'attribut "VALEUR" ou "COFACTEUR" .
- L'opposé d'une somme ("+" ou "-"), est la somme des opposés.
- L'opposé d'un produit ("*" ou "/") est le produit de, l'opposé d'un des facteur, par l'autre facteur.
- L'opposé d'une <ID_FONC> s'obtient, en multipliant son cofacteur par "-1".
- L'opposé d'une puissance "<CPUISS>", s'obtient en multipliant l'arbre par "-1".

On voit que l'algorithme se termine puisque, d'une part l'opposé d'un arbre est défini en fonction de ses sous-arbres et d'autre part l'opposé des feuilles pendantes s'obtient directement.

b) On peut raisonner de manière similaire pour la suppression de l'opérateur "/". Nous pourrions le remplacer par "*" si l'on construisait l'inverse de la branche de droite. "Trouver l'inverse d'un arbre" se défini similairement à "trouver l'opposé", de manière récursive:

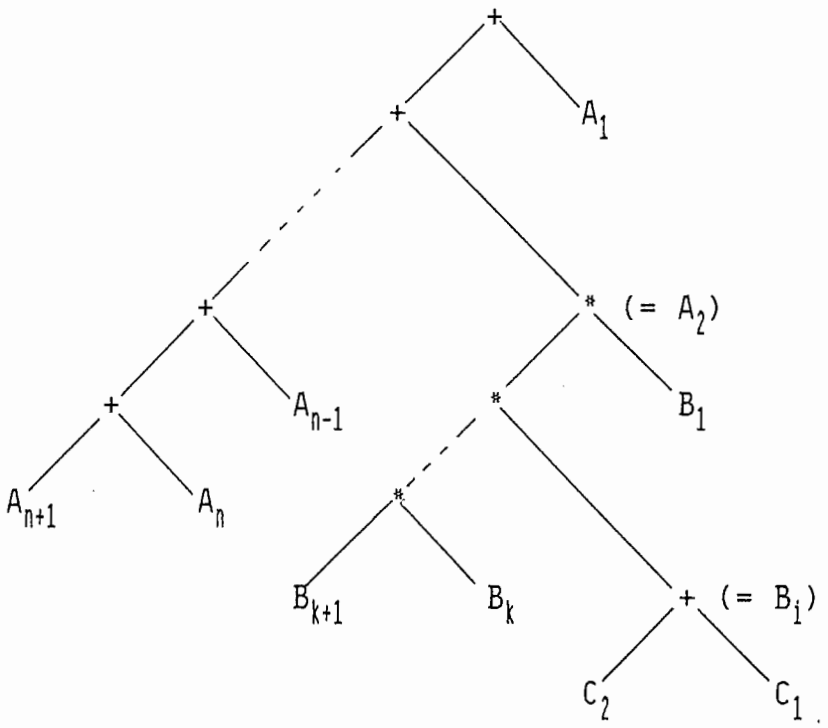
- L'inverse des feuilles pendantes ("<RAT>" ou "<VARIABLE>") s'obtient directement en modifiant leurs attributs valués.
- L'inverse d'une multiplication s'obtient en inversant chaque facteur.
- L'inverse d'une division s'obtient en intervertissant les deux sous-arbres du noeud.

- L'inverse d'une puissance s'obtient en créant l"opposé de la branche de droite.
- L'inverse d'une "<ID_FONC"> s'obtient en inversant son cofacteur et en multipliant son attribut "PUISS" par "-1".
- L'inverse d'une somme ("+" ou "-") s'obtient en élevant l'arbre à la puissance "-1", en créant un nouveau noeud.
- L'opposé d'une puissance "<CPUISS>", s'obtient en multipliant l'arbre par "-1".

Tout comme pour "l'opposé", l'algorithme se termine puisque, d'une part l'inverse d'un arbre est défini en fonction de ses sous-arbres et d'autre part l'inverse des feuilles pendantes s'obtient directement.

c) Maintenant que avons créé un arbre algébriquement identique à l'arbre de départ sans noeud de type "<CMOINS>" et "<CFOIS>", nous pouvons reconsidérer notre "normalisation des arbre" vue au paragraphe précédent. Nous pouvons facilement ramener n'importe quels arbres à un arbre dont chaque noeud de type "+" et "*" (il n'y a plus "-" et "/") n'aurait pas comme branche de droite, un arbre dont la racine lui serait identique.

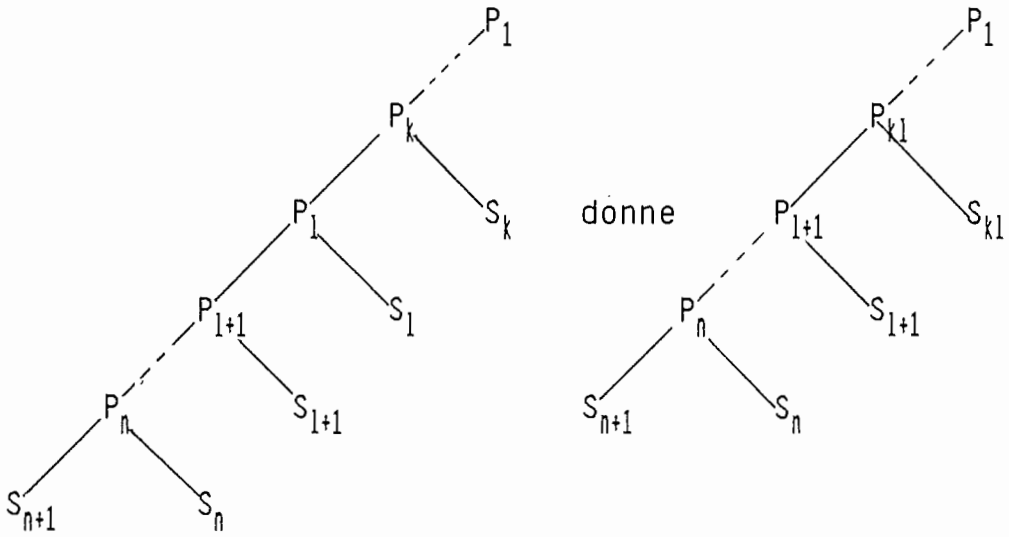
ex :



d) On pourrait permuter entre elles les branches "A_i" ainsi que les branches "B_i" et "C_i" de telle sorte que les suites "A_1, A_2, ..., A_{n+1}", "B_1, B_2,..., B_k" et "C_1, C_2" soient ordonnées. Cet arbre serait toujours sémantiquement identique à l'arbre de départ.

e) Dans une telle suite ordonnée "S_1, S_2,..., S_n" relative à une suite de "n-1" opérateurs "P". Deux éléments "S_k" et "S_l" pouvant donner lieu à une simplification, seront nécessairement contigus. On peut les regrouper en un seul élément "S_{kl}" et supprimer le noeud devenu inutile.

ex:



4) CONCLUSIONS

Il est maintenant possible de dire que toutes les expressions, sémantiquement identiques, obtenues par permutation des opérandes relatives aux opérateurs "+","-","*","/" peuvent avoir pour représentation interne un arbre unique.

De-plus, cet arbre sera simplifié, puisque que l'on aura opérer les regroupements et les simplifications des éléments similaires, présent dans l'expression de départ.

Il est possible, également, de mettre à profit les propriétés des élément neutre et absorbants relatif à chaque opérateur

IV. LA DERIVATION

1) INTRODUCTION

La dérivation des fonctions d'une variable réelle n'est pas un problème difficile en soi. Elle peut très bien se ramener à l'application systématique de règles qui laisse très peu de liberté dans le choix des étapes. La complexité du processus dépendra d'une part de la représentation des fonctions à dériver et d'autre part et de la définition du processus de dérivation, lui-même.

2) REGLES DE DERIVATION

Nous énumérons ici les règles de dérivation que nous utiliserons:

a) $(F(x) + V(x))' ::= F'(x) + V'(x)$

utilisable pour la dérivation des noeuds de type: <CPLUS>

b) $(F(x) - V(x))' ::= F'(x) - V'(x)$

utilisable pour la dérivation des noeuds de type: <CMOINS>

c) $(F(x) * V(x))' ::= (F'(x) * V'(x)) + (V'(x) * F(x))$

utilisable pour la dérivation des noeuds de type: <CFOIS>

d) $(F(x) / V(x))' ::= \frac{(F'(x) * V'(x)) - (V'(x) * F(x))}{V(x) \hat{} 2}$

utilisable pour la dérivation des noeuds de type: <CDIV>

e) $(F(x) \hat{} V(x))' ::= (F(x) \hat{} V(x)) * (V(x) * LN(U(x))$

utilisable pour la dérivation des noeuds de type: <CEXP>

f) (F(V(x)))' ::= F'(V(x)) * V'(x)

utilisable pour la dérivation des noeuds de type: <ID_FONC> pour lesquels:

- l'attribut "PUISS" serait différent de l'unité
- l'attribut "SUITE" différent de "x"

g) SIN'(x) ::= COS (x)

utilisable pour la dérivation des noeuds de type: <ID_FONC> pour lesquels "NOMF" vaut "SIN".

h) COS'(x) ::= - SIN (x)

utilisable pour la dérivation des noeuds de type: <ID_FONC> pour lesquels "NOMF" vaut "COS".

i) LN'(x) ::= 1/x

utilisable pour la dérivation des noeuds de type: <ID_FONC> pour lesquels "NOMF" vaut "LN".

j) EXP'(x) ::= EXP (x)

utilisable pour la dérivation des noeuds de type: <ID_FONC> pour lesquels "NOMF" vaut "EXP".

k) ASIN'(x) ::= $\frac{1}{(1 - x^2)^{1/2}}$

utilisable pour la dérivation des noeuds de type: <ID_FONC> pour lesquels "NOMF" vaut "ASIN".

l) ACOS'(x) ::= $\frac{-1}{(1 - x^2)^{1/2}}$

utilisable pour la dérivation des noeuds de type: <ID_FONC> pour lesquels "NOMF" vaut "ACOS".

m) ATG'(x) ::= $\frac{1}{1 + x^2}$

utilisable pour la dérivation des noeuds de type: <ID_FONC> pour lesquels "NOMF" vaut "ATG".

n) ACOTG'(x) ::= $\frac{1}{1 + x^2}$

utilisable pour la dérivation des noeuds de type: <ID_FONC> pour lesquels "NOMF" vaut "ACOTG".

o) a' ::= Ø

utilisable pour la dérivation des noeuds de type: <RAT>

p) (a * x^b)' ::= (a * b) * x^(c-1)

utilisable pour la dérivation des noeuds de type: <VARIABLE>

3) REMARQUES

a) Il est remarquable que: *à chaque type de noeud de nos arbres algébriques ne corresponde qu'une et une seule règle de dérivation.*

b) Il est tout aussi remarquable que: *une règle relative à un noeud ne mette en jeu que les sous-arbres de ce noeud.*

c) Les règles de dérivation sont définies récursivement.

d) Les règles relatives aux noeuds pouvant constituer une feuille pendante (terminale) d'un arbre (<RAT> et <VARIABLE>) ne font pas intervenir la récursivité

e) Elles ne font intervenir que des opérations définies sur nos arbres. Leur mise en oeuvre sur arbre algébrique donnent donc naissance à un arbre algébrique

f) Des trois propriétés "a","b" et "c" découle l'algorithme de dérivation: Il s'agira d'un algorithme récursif qui, appliqué au noeud formant la racine d'un arbre, retournera un nouvel arbre qui sera la représentation de la dérivée de l'arbre de départ.
A chaque règle de dérivation sera associée une routine sémantique qui implémente la construction du nouvel arbre.

g) Les règles "o" et "p" constituent des redondance de la règle "e", mais elles valident la propriété énoncée au point "a" et rendent l'algorithme plus efficace.

h) Les règles "b" et "d" constituent une preuve de terminaison de l'algorithme. Celui-ci atteindra toujours sa condition d'arrêt; d'une part la progression est strictement monotone (règle "b"); et d'autre part la récursivité prend fin aux feuilles pendantes.

4) CONCLUSIONS

Toutes les propriétés de notre algorithme de dérivation justifie à posteriori certains points de notre grammaire et de nos structures de données:

- Notamment, le choix délibéré d'adjoindre aux éléments de type <VARIABLE> et <ID_FONC>, les attributs "COFACTEUR" et "PUISS" car sans ceux-ci, nous n'aurions pu nous donner la règle de dérivation "p", qui assure, avec la règle "o" la preuve de terminaison de notre algorithme. Nous aurions du également faire sans cesse usage de la règle "e", ce qui aurait compliqué les mécanismes de simplification et diminué l'efficacité du programme.

- Nous aurions également rencontré certaines difficultés eu égard à la règle de dérivation "c", si nous avions choisi d'implémenter des arbres n-aires pour mieux transcrire la notion de produit.

En fait, cet algorithme de dérivation à été le point de départ de notre analyse et c'est lui qui nous a guidé dans nos choix de représentation. La question était alors "Comment implémenter la dérivation sur ordinateur ?"

DETAILS D'IMPLEMENTATION

I. INTRODUCTION

Ce chapitre se veut être un guide au lecteur qui désirerait comprendre, utiliser voire modifier nos procédures. Il ne représente pas une analyse de notre implémentation mais plutôt un support pour une meilleur perception de notre texte "Turbo Pascal". Nous ne serons pas exhaustif, les textes "Pascal" étant suffisamment explicites.

Nous mettons néanmoins en garde le lecteur qui ne se serait pas attardé sur le chapitre précédent, sur le fait que, une bonne connaissance de ce dernier est indispensable à la compréhension des mécanisme de nos programmes.
Nous essaierons d'établir des relations avec le chapitre précédent en renvoyant le lecteur aux concepts développés précédemment.

II. TYPES ET VARIABLES

1) LES TYPES (listing p.2)

a) PRATIONNEL

Dans le souci de garder une représentation des nombre aussi usuelle que possible, nous avons voulu nous donner la possibilité d'écrire certains rationnels sous forme de fraction. Dans le cadre d'un didactitiel, on préférera l'écriture "1/3" à "0.33333333E00".
C'est ce que nous réalisons avec le type *"PRATIONNEL"*

- Ils représentent tantôt un rationnel "q" sous forme fractionnaire, c'est-à-dire d'un quotient d'un entier par un naturel (q = z/n), tantôt un réel quelconque.
- Nous prenons pour convention que la représentation d'un nombre non exprimable sous forme de fraction aura un dénominateur négatif.

- Nous abandonnerons la représentation sous forme de fraction pour les rationnels dont le dénominateur serait supérieur à 181.

b) ARBRE_SYMBOL

Ce type nous donne l'ensemble des classes syntaxiques de notre grammaire. (voir p. 17)

c) FONCTION

Ce type nous donne l'ensemble des fonctions mathématiques dont peuvent faire usage nos expressions.

d) ARBRE

L'objet évoqué par ce type est l'élément de base de la construction de nos arbres algébriques. Ils correspondent à la syntaxe valuée de notre grammaire:

- **NOM1**: représentes la classe syntaxique de l'élément et ne sert qu'à la phase de l'analyse syntaxique. Une fois l'analyse syntaxique terminée, il n'aura plus la moindre utilité.
- **NOM2**: donne le symbole terminal dont est issu l'élément. Il est vital, car c'est lui qui caractérise chaque noeud. il interviendra dans toutes les procédures et fonctions
- Nous regrettons ici de ne pas avoir adopté une structure identique pour les différentes familles de noeuds: Les noeuds auront des **attributs** distincts portant des désignations différentes selon la valeur de **NOM2**:
  - s'il s'agit d'un **opérateur** (binaire): deux *pointeurs* dirigés vers les opérandes gauche et droite.

- s'il s'agit d'une **fonction** ou d'une **variable**:
  - deux *PRATIONNEL* donnant la valeur du cofacteur et de la puissance associés à l'élément
  - non utilisés pour les variables mais néanmoins existants: un identificateur de *FONCTION* et un *pointeur* marquant l'arbre auquel s'applique la fonction
- s'il s'agit d'un nombre (<RAT>): un *PRATIONNEL*

d) ARBRE2

Certaines procédures travaillant sur le type *arbre* ont besoin d'une variable *string* dans leur liste d'appel pour retourner un éventuel message d'erreur. Nous avons parfois préféré par la suite, l'utilisation d'une variable globale. Ceci constitue un point à revoir.

2) CONSTANTES ET VARIABLES (listing p.2)

a) SRE

Ce tableau implémente la table S.R.E. réduite; c'est-à-dire, que dans la table S.R.E étendue (voir p.25), seules les colonnes correspondant aux symboles terminaux sont utiles à l'algorithme.

b) LIST_FONC

Contient les patterns interprétés par l'analyseur lexicographique comme étant des identificateurs de fonction.

c) LIST_NR_FONC

fait correspondre à chaque élément de *LIST_FONC* un élément de type *FONCTION*.

d) STRERROR et ERREUR

Ces deux variables globales sont utilisées lorsqu'une erreur non recouvrable se produit. STRERROR contient alors une explication de l'erreur et ERREUR prend la valeur "TRUE".

III. FONCTIONS MATHEMATIQUES

1) FONCTIONS D'UNE VARIABLE REELLE

Quelques fonctions circulaires sur les Réels inexistantes en *Turbo Pascal* ont du être implémentées, notamment:

```
function asin(x:real):real;(listing p.2)
function acos(x:real):real;(listing p.2-3)
function tg(x:real):real;(listing p.3)
function cotg(x:real):real;(listing p.3)
function acotg(x:real):real;(listing p.3)
```

2) FONCTIONS D'UNE VARIABLE FRACTIONNAIRE ET/OU REELLE

Notre choix de garder tant que possible, une représentation fractionnaire des nombres rationnels, nous a amené à implémenter ou réimplémenter certaines fonctions de base. Beaucoup de ces fonctions ou procédures manipulent des *"PRATIONNEL"*. au dénominateur). Nous ne gardons sous forme fractionnaire que les Rationnels formés du quotient de deux entiers "I_1/I_2" dont le dénominateur est inférieur à 181 (I_2<181).

a) function pgcd (un,deux:real):real (listing p.3-4)

Cette fonction retourne le Plus Grand Commun Diviseur des entiers *un* et *deux* passés en entrée. Nous travaillons avec le Type *Real* pour des raisons de compatibilité avec d'autres procédures.

L'algorithme est basé sur la recherche de nombres premiers donnant un quotient entier pour chacun des deux nombres. Cette recherche de *PGCD* ne va pas au-delà de 181.

b) procedure simplifie (un:prationnel) (listing p.4)

Cette procédure divise le numérateur et le dénominateur par leur *PGCD* s'il s'agit d'un nombre fractionnaire.
On pourrait y adjoindre un algorithme d'essai de mise des réels (dénominateur = -1) sous forme fractionnaire.

c) Les procédures "Plus, Moins, Fois, Divis et Puissances" (listing p.4-7)

Ces procédures réimplémentent pour les *"Prationnels"* les opérateurs classiques "+, -, *, / et ^".
Elles reçoivent trois pointeurs en entrée: deux pointant vers les opérandes et un, vers le résultat.

Elles essaient de fournir un résultat sous forme fractionnaire.

d) procedure inverser rat(entree:prationnel); (listing p.7-8)

Cette procédure inverse le Prationnel reçu en entrée. Si celui-ci est représenté sous forme fractionnaire, il suffit d'échanger numérateur et dénominateur. Notons que l'éventuel signe négatif doit rester au numérateur.

IV. PROCEDURES ET FONCTIONS DE MANIPULATION D'ARBRES

1) procedure effacer arbre(entree:arbre); (listing p.9)

Cette procédure libère l'espace mémoire occupé par l'arbre passé en entrée. Il procède de façon récursive en partant des feuilles pour remonter vers la racine.

2) function var to rat(entree:arbre):arbre; (listing p.8)

Cette fonction a été rendue nécessaire par un choix regrettable dans la structure des données. Nous avons différentié par leur représentation les éléments de type "<RAT>" des éléments de type "<VARIABLE> ET <ID_FONC>", or, dans le cas où la puissance de ces deux derniers éléments est nulle, ils devraient devenir compatibles avec les éléments de type "<RAT>", ce qui n'est malheureusement pas le cas.

Cette fonction retourne un élément de type "<RAT>" dont la valeur est égale au cofacteur de l'élément de type "<VARIABLE> ET <ID_FONC>" passé en entrée et, efface ce dernier élément ou l'arbre dont il est racine.

3) function identique(un,deux:arbre):boolean; (listing p.8)

Cette fonction retourne le résultat de la comparaison des deux arbres passés en entrée. S'ils sont identiques, elle retourne *"TRUE"* sinon, *"FALSE"*.
Elle scrute les deux arbres récursivement.
La fonction ne repère malheureusement pas deux arbres tout à fait équivalents, s'il ne sont pas rigoureusement identiques. Comme nous le verrons plus loin, ceci ne constitue en rien un obstacle.

4) function lire arbre(entree:arbre;priorite:pbyte):string;
(listing p.9-12)

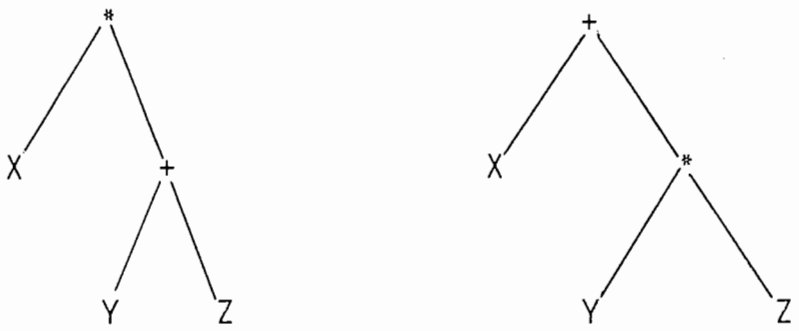
a) function rat to str (listing p.9-10)

Cette fonction retourne une chaîne de caractères correspondant au *Prationnel* passé en entrée.
Les entiers ne seront représentés que par leur valeur entière tandis que les nombres réels prendront le format scientifique.
Les nombres fractionnaires seront encadrés par des parenthèses.

Les paramètres passés en entrée sont:
- un pointeur vers l'arbre qui est à lire;
- la priorité de l'opérateur père;
  cette priorité vaut "1" pour les opérateurs de somme "+" et "-", "2" pour les opérateurs de produit "*" et "/";
- la position de l'arbre par rapport à cet opérateur; c'est-à-dire s'il en constitue l'opérande gauche ou l'opérande droite?

b) Corps de la fonction lire arbre (listing p.10-12)
Cette fonction lit récursivement l'arbre passé en entrée.
Les priorités permettent de sortir des expressions non complètement parenthétisées:
Si l'on se trouve sur un noeud correspondant à un opérateur binaire dont la priorité est strictement inférieure à celle de l'opérateur appelant alors, la lecture de l'arbre doit être encadrée de parenthèses.

ex:



donne:

X * (Y + Z) X + Y * Z

Si la priorité est supérieure à 4 alors, l'arbre à lire est une opérande droite de l'opérateur appelant sinon, il s'agit d'une opérande gauche.

Pour éviter des écriture comme X + -2" au lieu de "X - 2" ou "cos(X) * X^-2" au lieu de "cos(X) / X^2", un flag peut être transmis à la procédure appelante pour changer éventuellement d'opérateur:

- Si il s'agit d'une opérande droite
  et si l'opérateur appelant est de priorité "1" ("+","-")
  et si la lecture de cet opérande commence par un "-",
  alors:
  La chaîne de caractère perd ce "-" et transmet un flag à la procédure appelante qui changera l'opérateur "-" en "+", ou l'opérateur "-" en "+".

- Si il s'agit d'une opérande droite
  et si l'opérateur appelant est de priorité "2" ("*","/")

et si un des facteurs constituant cette opérande a une puissance négative,

alors:

Cette puissance change de signe et un flag est transmis à la procédure appelante qui changera l'opérateur "*" en "/", ou l'opérateur "/" en "/".

Cette stratégie n'a été adoptée que pour les arbres situés à droite des opérateurs binaires, les arbres situés à gauche gardant leur éventuel signe "-" ou leur puissance négative.
Puisque les opérateurs "somme(+)" et "produits(*)" sont commutatifs, une amélioration consistant à permuter deux opérandes lorsque celle de gauche est négative ou de puissance négative, pourrait facilement être implémentée, mais l'ordre dans lequel apparaissent les termes d'une somme et les facteurs d'un produit correspond à une stratégie expliquée ailleurs.

5) function calcul arbre(pommier:arbre;x:real):real;
(listing p.13-14)

Cette fonction retourne la valeur réelle correspondant à l'évaluation algébrique de l'arbre "pommier", passé en entrée, en particularisant la variable "X" de cet arbre par la valeur réelle "X" passée également en entrée.
L'évaluation se fait récursivement, des feuilles de l'arbre vers la racine.

6) PROCEDURE do arbre(donnee:arbre2); (listing p.14-31)

Cette procédure est capitale. Elle se charge de la transposition d'une expression, de sa représentation sous forme de caractères vers sa représentation sous forme d'arbre algé-

brique. Elle implémente le passage de la représentation externe vers la représentation interne.

La compréhension de son fonctionnement requiert une parfaite maîtrise du point II. du 1er chapitre (voir p.27-29).
Nous ne pouvons ici redévelopper tous les concepts et mécanismes auxquels nous avons recours. Au contraire, nous essaierons de faire un parallélisme entre ce qui à été dit à propos de l'algorithme précédemment développé, et notre procédure.

A) PROCEDURE SHIFT: (listing p.14-19)

Pour les raisons précisées (au point e) p.22), elle intègre *l'analyse lexicographique* et *l'action shift* proprement dite qui ne concerne, en fait, qu'une dizaine de lignes de programme. La tâche essentielle de la procédure est donc l'analyse lexicographique.
Rappelons que cette dernière a pour finalité, d'extraire du ruban de caractères initial *"entree"*, un pattern compatible avec un symbole terminal de notre grammaire et d'en produire un élément de type *arbre*.

Nous remarquerons que les caractères utilisés pour la représentation externe de chaque symbole terminal forment, à peu de choses près, des ensembles disjoints. Seuls les identificateurs de fonctions et les variables, utilisant le même alphabet (a..z,A..Z) enfreignent la règle.
Cette exception aurait pu être contournée pour simplifier notre tâche:

- nous aurions pu imposer à l'utilisateur de ne pas nommer ses variables en se servant comme première lettre, des premières lettres des identificateurs de fonctions que nous utilisons à savoir: *"a,c,e,l,s,t,A,C,E,L,S,T"* (voir LIST_FONC, listing p.1).
- La procédure CAS_MOT permet de lever l'ambiguïté.

Une fois obtenu l'équivalent syntaxique des morceaux de chaînes de caractères, il faut encore les valuer:

- Il faut pouvoir convertir les séquences de chiffres (et de "." ou ",") en nombre; ce que fait la procédure CAS_NOMBRE.
- Il faut aussi pouvoir transformer le string représentant un identificateur de fonction en nom de fonction; voir la procédure CAS_MOT.

a) PROCEDURE cas variable(chaine:STRING); (listing p.14-15)

Cette procédure crée un élément <VARIABLE> et lui value ses attributs "*cofacteur*" et "*puiss*" à "1".

Si la "*chaîne*" a pour valeur "*PI*" ou "*E*", elle crée un élément <RAT> qu'elle value à la valeur correspondante.

Tous les autres patterns donneront naissance à la classe syntaxique "<VARIABLE>". Ceci doit être corrigé: il faut adjoindre à la procédure une mémoire pour refuser des expressions du genre " X + Y " qui est actuellement équivalent dans le programme à " X + X ". Ceci constitue une modification mineure.

b) PROCEDURE cas nombre; (listing p.15-16)

Cette procédure teste l'absence de double virgule au sein de la séquence de chiffres, crée un élément <RAT> et le value à la valeur correspondante.

c) PROCEDURE cas mot; (listing p.16-17)

Elle construit un pattern formé de la succession des caractères appartenant à (a..z,A..Z) et teste si le pattern constitue un identificateur de fonction; si

oui, il crée un élément de type <ID_FONC> sinon, elle appelle la procédure CAS_VARIABLE.

B) PROCEDURE REDUCE: (listing p.19-30)

Elle réalise l'action REDUCTION de notre algorithme (voir p.20, 28-29).
Les différentes procédures "REDUCE_i" réalisent l'application des règles de production auxquelles elles s'accordent.

a) FUNCTION PRODUCTION (listing p.19-20)

Cette fonction procède à une série de tests pour retourner le numéro de la règle de production qu'il convient d'appliquer, à ce stade de la traduction, aux éléments formant le sommet de la pile.

Si aucune de ces règles ne convient, c'est que l'expression est syntaxiquement non correcte.

b) PROCEDURE reduce_1: (listing p.20-21)

Elle correspond à la règle de production:
"<express>:=<express><+><terme>".

Elle crée un nouvel arbre de classe syntaxique <EXPRESS> dont les attributs seront établis en fonction de la nature des deux éléments <EXPRESS> et <TERME> initiaux:

- S'il sont compatibles par rapport à l'addition, c'est qu'il sont identiques, à un facteur près, et le nouveau noeud sera de même type que les deux autres.
  Ceci est réalisé lorsque l'attribut "**NOM2**" des deux éléments en présence sont, par exemple:
  - deux <RAT>;
  - deux <VARIABLE> de même puissance;

- deux <ID_FONC> de même puissance et dont les attributs "**NOMF**" et "**SUITE**" sont strictement identiques.
- Sinon il y a édification d'un nouvel arbre dont l'élément racine sera le symbole terminal <CPLUS>, les deux éléments originels devenant, tel quels, les branches gauche et droite du nouveau noeud.

c) PROCEDURE reduce_2; (listing p.21-22)

Elle correspond à la règle de production:
"<express>:=<express><-><terme>".

L'addition et la soustraction vérifiant, outre la commutativité, les mêmes propriétés, la construction du nouvel arbre est tout à fait similaire à la construction relative à la procédure REDUCE_1.

d) PROCEDURE reduce_3; (listing p.22)

Elle correspond à la règle de production:
"<express>:=<terme>".

Elle n'a pas de valeur sémantique et ne sert qu'à l'analyse syntaxique: elle donne à l'attribut "**NOM1**" de l'élément au sommet du stack, la valeur "<EXPESS>".

e) PROCEDURE reduce_4; (listing p.22-23)

Elle correspond à la règle de production:
"<terme>:=<terme><*><facteur>".

Elle crée un nouvel arbre de classe syntaxique <TERME> dont les attributs seront établis en fonction de la nature des deux éléments <TERME> et <FACTEUR> initiaux:

- S'il sont compatibles par rapport à la multiplication, le nouveau noeud sera obtenu par exécution immédiate du produit de ces deux éléments. Ceci est réaliser lorsque leurs attributs "**NOM2**" sont, par exemple:
  - deux <RAT>;
  - deux <VARIABLE>;
  - deux <ID_FONC> dont les attributs "**NOMF**" et "**SUITE**" sont strictement identiques.

  Ce produit est obtenu par multiplication des cofacteurs et addition des puissances.
- Sinon il y a édification d'un nouvel arbre dont l'élément racine sera le symbole terminal <CFOIS>, les deux éléments originels devenant, tel quels, les branches gauche et droite du nouveau noeud.

f) PROCEDURE reduce_5; (listing p.23-25)

Elle correspond à la règle de production:
"<terme>:=<terme></><facteur>".

La multiplication et la division vérifiant, outre la commutativité, les mêmes propriétés, la construction du nouvel arbre est tout à fait similaire à la construction relative à la procédure REDUCE_4.

g) PROCEDURE reduce_6; (listing p.25)

Elle correspond à la règle de production:
"<terme>:=<facteur>".

Elle n'a pas de valeur sémantique et ne sert qu'à l'analyse syntaxique: elle donne à l'attribut "**NOM1**" de l'élément au sommet du stack, la valeur "<TERME>".

h) PROCEDURE reduce_7; (listing p.25-26)

Elle correspond à la règle de production:
"<facteur>:=<facteur><^><cvaleur>".

Elle crée un nouvel arbre de classe syntaxique <FACTEUR> dont les attributs seront établis en fonction de la nature des deux éléments <FACTEUR> et <CVALEUR> initiaux:

- Si les attributs "**NOM2**" de ces deux éléments ont pour valeur:
  - "**<RAT>**" pour <CVALEUR>;
  - "**<RAT>**","**<VARIABLE>**" ou "**<ID_FONC>**", pour <FACTEUR>,

  alors, le nouveau noeud sera obtenu par la mise, immédiate, à la puissance du premier élément par le second et ce, en fonction de l'attribut "**NOM2**" de l'élément <FACTEUR>:
  - s'il s'agit d'un <RAT>, on applique la procédure PUISSANCE aux deux éléments;
  - s'il s'agit d'une <VARIABLE> ou d'un <ID_FONC>, on appliquera la procédure PUISSANCE aux deux cofacteurs et la procédure PLUS aux deux exposants.
- Sinon il y a édification d'un nouvel arbre dont l'élément racine sera le symbole terminal <CEXP>, les deux éléments originels devenant, tel quels, les branches gauche et droite du nouveau noeud.

i) PROCEDURE reduce_8; (listing p.26)

Elle correspond à la règle de production:
"<facteur>:=<cvaleur>".

Elle n'a pas de valeur sémantique et ne sert qu'à l'analyse syntaxique: elle donne à l'attribut "**NOM1**" de l'élément au sommet du stack, la valeur "<facteur>".

j) PROCEDURE reduce 9: (listing p.26)

Elle correspond à la règle de production: "<cvaleur>:=<(><express><)>".

Elle n'a pas de valeur sémantique et ne sert qu'à l'analyse syntaxique: elle donne à l'attribut "**NOM1**" du deuxième élément du stack, <EXPRESS> la valeur "<CVALEUR>" et supprime de la pile le premier et le troisième élément(<)> et <(>).

k) PROCEDURE reduce 10: (listing p.26-27)

Elle correspond à la règle de production: "<cvaleur>:=<->><(><express><)>".

Il s'agit, ici de créer un arbre qui soit l'opposé par rapport à l'unité, de l'arbre représenté par l'élément <EXPRESS>. Tous les cas de figure doivent être pris en compte en fonction de la nature de l'attribut "**NOM2**" de ce noeud:

- S'il s'agit d'un "<RAT>",d'un "<ID_FONC>" ou d'un "<VARIABLE>", trouver son opposé ne pose aucun problème.
- S'il s'agit d'un produit ("*", "/"), il suffit d'opposer l'un des deux facteurs; le premier par exemple.
- S'il s'agit d'une somme ("+" et "-"), il convient d'opposer les deux termes.
- S'il s'agit de l'opérateur puissance ("^"), il faut multiplier l'arbre par "-1".

l) PROCEDURE reduce 11; (listing p.27)

Elle correspond à la règle de production:
"<cvaleur>:=<fonc>".

Elle n'a pas de valeur sémantique et ne sert qu'à l'analyse syntaxique: elle donne à l'attribut "**NOM1**" de l'élément au sommet du stack, la valeur "<CVALEUR>".

m) PROCEDURE reduce 12; (listing p.27-28)

Elle correspond à la règle de production:
"<cvaleur>:=<->fonc>".

Elle constitue un cas particulier de la procédure REDUCE_10.

n) PROCEDURE reduce 13; (listing p.28)

Elle correspond à la règle de production:
"<cvaleur>:=<rat>".

Elle n'a pas de valeur sémantique et ne sert qu'à l'analyse syntaxique: elle donne à l'attribut "**NOM1**" de l'élément au sommet du stack, la valeur "<CVALEUR>".

o) PROCEDURE reduce 14; (listing p.28)

Elle correspond à la règle de production:
"<cvaleur>:=<->rat>".

Elle constitue un cas particulier de la procédure REDUCE_10.

p) PROCEDURE reduce 15; (listing p.28)

Elle correspond à la règle de production:
"<cvaleur>:=<variable>".

Elle n'a pas de valeur sémantique et ne sert qu'à l'analyse syntaxique: elle donne à l'attribut "**NOM1**" de l'élément au sommet du stack, la valeur "<CVALEUR>".

q) PROCEDURE reduce 16; (listing p.28)

Elle correspond à la règle de production:
"<cvaleur>:=<->variable>".

Elle constitue un cas particulier de la procédure REDUCE_10.

r) PROCEDURE reduce 17; (listing p.28-29)

Elle correspond à la règle de production:
"<fonc>:=<id_fonc><cvaleur>".

Elle donne à l'attribut "**NOM1**" de l'élément <ID_FONC>, la valeur <FONC> et à l'attribut "SUITE" la valeur du pointeur dirigé vers <CVALEUR>.

Dans le cas particulier ou <CVALEUR> correspond à un <RAT>, elle crée un nom dont l'attribut "**NOM2**" prendra la valeur <RAT> et l'attribut <VALEUR>, la valeur retournée par l'évaluation de la fonction.

Elle crée un nouvel arbre de classe syntaxique <FONC> dont les attributs seront établis en fonction de la nature des deux éléments <FACTEUR> et <CVALEUR> initiaux:

- Si les attributs "**NOM2**" ont pour valeur:
  - "**<RAT>**" pour le second;
  - "**<RAT>**","**<VARIABLE>**" ou "**<ID_FONC>**", pour le premier,

  alors, le nouveau noeud sera obtenu par la mise à la puissance immédiate du premier élément par le second, en fonction de l'attribut "**NOM2**" du premier:
  - s'il s'agit d'un <RAT>, on applique la procédure PUISSANCE aux deux éléments;
  - s'il s'agit d'une <VARIABLE> ou d'un <ID_FONC>, on appliquera la procédure PUISSANCE aux deux cofacteurs et la procédure PLUS aux deux exposants.
- Sinon il y a édification d'un nouvel arbre dont l'élément racine sera le symbole terminal <CEXP>, les deux éléments originels devenant, tel quels les branches gauche et droite du nouveau noeud.

Toutes ces procédures sont assez lourdes et ont perdu en partie leurs raisons d'être, après la réalisation des routines de simplification d'arbres: "OPPOSER, INVERSER et REDUIRE_ARBRE". Il y a donc moyen de simplifier notre implémentation, mais nous perdrions leur relative indépendance sans avoir la certitude de gagner en efficacité.

7) function copy_arbre(poirier:arbre):arbre; (listing p.32-33)

Cette fonction retourne un pointeur vers une copie de l'arbre passé en entrée.

8) function opposer(entree:arbre):arbre; (listing p.33)

Dans le but de supprimer de l'arbre l'opérateur non commutatif "-" et de le remplacer par l'opérateur "+", il était nécessaire d'obtenir de manière simple l'opposé d'un arbre;

c'est-à-dire, en terme algébrique, son complémentaire par rapport à l'élément neutre "0". Tous les cas de figures doivent être pris en compte:

- S'il s'agit d'un "<RAT>",d'un "<ID_FONC>" ou d'un "<VARIABLE>", trouver son opposé ne pose aucun problème.
- S'il s'agit d'un produit ("*", "/"), il suffit d'opposer l'un des deux facteurs; le premier par exemple.
- S'il s'agit d'une somme ("+" et "-"), il convient d'opposer les deux termes.
- S'il s'agit de l'opérateur puissance ("^"), il faut multiplier l'arbre par "-1".

9) function inverser(entree:arbre):arbre; (listing p.34)

Dans le but de supprimer de l'arbre l'opérateur non commutatif "/" et de le remplacer par l'opérateur "*", il était nécessaire d'obtenir de manière simple l'inverse d'un arbre par rapport à l'unité. Tous les cas de figure doivent être pris en compte:

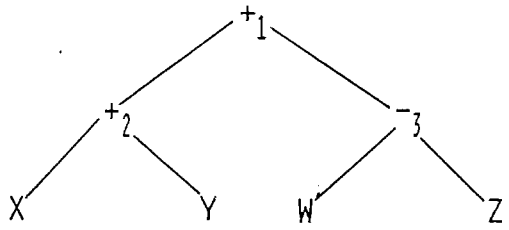
- S'il s'agit d'un "<RAT>" trouver son opposé ne pose aucun problème.
- S'il s'agit d'un "<ID_FONC>" ou d'un "<VARIABLE>", il faut inverser leur cofacteur et opposer leur puissance.
- S'il s'agit d'un produit ("*"), il suffit d'*inverser* (récursivement) les deux facteurs.
- S'il s'agit d'une division, il suffit d'intervertir les deux opérandes.
- S'il s'agit d'une somme ("+" et "-"), il convient de retourner la puissance "-1" de cette somme.
- S'il s'agit de l'opérateur puissance ("^"), il faut *opposer* la deuxième opérande.

10) function rectifier(poirier:arbre;nom:arbre symbol):arbre;
(listing p.34-35)

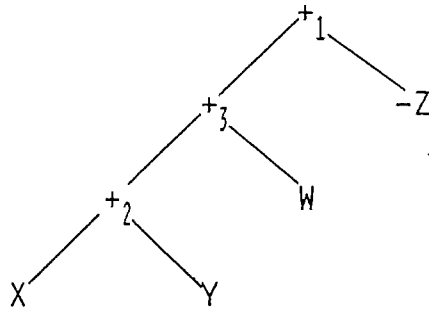
Cette fonction réalise un point essentielle de la normalisation des arbres. Il a fallu pallier à notre choix d'implémentation d'arbres binaires qui enlevait la notion de somme et de produit algébrique, très bien matérialisée par les arbre n-aires. La commutativité est difficile à implémenter sur les arbres binaire en général, à moins qu'ils se présentent sous une forme très particulière que nous appellerons *"Forme Normale"* qui consiste en:

- La suppression des deux opérateurs non commutatifs "-" et "/".
- Ramener à gauche des noeuds matérialisant un opérateur commutatif ("+" ou "*"), les opérandes droites dont le noeud racine matérialiserait un opérateur de même priorité.

ex: (avec l'opérateur "+")



devient:



Présentés sous cette forme, les arbres offrent plus de facilité de tri. Il devient aussi plus facile de tirer parti de la commutativité des opérateurs. On voit que sous

la forme normale, les places de X, Y, Z et W sont tout à fait équivalentes et peuvent être interverties.

L'algorithme de cette fonction use de la récursivité, ainsi, les sous-arbres de droite (dont le noeud racine est "$-_3$" dans notre exemple) seront mis sous forme normale avant d'être insérés entre le noeud père et le noeud fils de gauche ("$+_1$" et "$+_2$" dans l'ex.) qui aura été préalablement mis, également, sous forme normale.

Le passage de l'opérateur "-" à l'opérateur "+" ("$-_3$" => "$+_3$" dans l'ex.) utilise la fonction "*opposer*" tandis que le passage de l'opérateur "/" à l'opérateur "*" utilise la fonction "*inverser*".

11) function trie arbre(entree:arbre):arbre; (listing p.35-37)

a) procedure effacer chaîne(entree:liste feuille); (listing p.35)

Libère la place occupée par la variable de type liste_feuille pointée par "entrée".

b) function do chaîne(poirier:arbre;nom:arbre symbol): liste feuille; (listing p.36)

Cette fonction retourne une liste chaînée de tous les noeuds de type "non" (<CPLUS> ou <CMOINS>) formant successivement les branche de gauche de l'arbre "poirier", passé en entrée.
Cette liste chaînée rassemble tous les noeuds sur lesquels on pourra opérer un tri.

c) procedure ordonne(chaine:liste feuille); (listing p.36-37)

Cette procédure est un "bubble sort" amélioré qui, lors de chaque passage opère sa phase de tri entre le

premier et le dernier élément intervertis lors du passage précédent.
Ce tri sera effectué sur les différents éléments rassemblés par la fonction "do_chaîne".

d) <u>function Intervertir(un,deux:arbre):boolean;</u> (listing p.36)

Cette fonction intervertit éventuellement les deux éléments "un" et "deux" passés en entrée, dans quel cas elle retourne "true" sinon elle retourne "false".

L'ordre fixé entre les différents éléments a intervertir est le suivant:

1) Si les "un" et "deux" appartiennent à {<RAT>, <VARIABLE>,<ID_FONC>} l'ordre est l'inverse de celui défini sur ces éléments.
2) Si les deux éléments sont des <ID_FONC>, l'ordre sera l'inverse de celui défini sur les noms de fonction (*TYPE fonction*)
3) Si les deux éléments sont de même type et appartiennent à {<VARIABLE>,<ID_FONC>}, l'ordre est l'inverse de leur puissance.
4) Les autres éléments, de structure plus complexe, sont systématiquement intervertis sauf s'il se ressemblent, pour qu'à la fin du tri, ils soient toujours proche l'un de l'autre.

e) <u>Corps de la fonction trie arbre</u> (listing p.37)

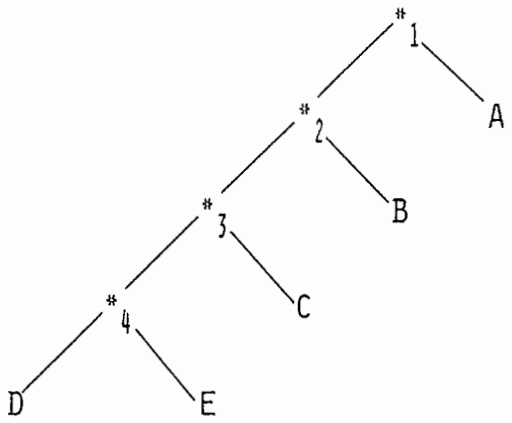
La fonction "Trie_arbre" séquence les trois procédures et fonctions qui précèdent et retourne un arbre algébriquement identique à celui passé en entrée mais, ordonné.

12) function reduire arbre(pommier:arbre):arbre; (listing p.37-43)

a) function elaguer(entrée:arbre;nom:arbre symbole)
(listing p.38-43)

Cette fonction reçoit en entrée un arbre supposé être sous sa forme normale et trié. Si des éléments peuvent se simplifier l'un l'autre, ils sont donc, après le tri, contigus dans l'arborescence.

ex:



A peut éventuellement se simplifier avec B qui peut lui-même éventuellement se simplifier avec C. Si A se simplifie avec B, {A,B} devient A', le noeud "$*_2$" disparaît pour laisser sa place au noeud "$*_3$" et on réessaye une simplification {A', C}.
S'il n'y a pas eu de simplification {A, B} alors on essaie la simplification suivante {B, C}, et ainsi de suite pour arriver à la dernière simplification {D, E}, un peu différente par la position relative des éléments.

Le "nom", passé en entrée peut prendre la valeur <CPLUS> ou <CFOIS> et nous renseigne sur le type d'arborescence dans laquelle nous nous trouvons.

Dans le cas d'une arborescence "PRODUIT", tous les cofacteurs prennent la valeur "1/1", sauf un (le

premier) qui prend pour valeur, le produit de tous les cofacteurs originaux.

b) Corps de la fonction reduire arbre (listing p.43)
Cette fonction retourne un arbre normalisé (voir p.31-33) et constitue ainsi l'étape essentielle de la simplification des arbres

Elle fonctionne récursivement et, si la racine de l'arbre "pommier", passé en entrée, est un opérateur binaire (<CPLUS>,<CMOINS>,<CFOIS>,<CDIV>,<CEXP>), elle s'applique elle-même à ses opérandes gauche et droite. Si la racine de "pommier" est de type "<ID_FONC>" elle s'applique à sa branche "SUITE". Les sous arbres de "pommier" sont donc sous forme normale.

Elle appelle les fonction RECTIFIER et ELAGUER pour mettre l'arbre sous sa forme normale. Les opérateurs non commutatifs "<CMOINS> et <CDIV>" sont donc éliminés et, s'il existe une opérande de type "<RAT>" elle se trouve nécessairement à droite.

Elle détruit éventuellement le noeud racine, s'il représente un noeud devenu inutile:
- l'opérateur "<CPLUS>" dont le terme de droite serait nul sera remplacé par le terme de gauche;
- l'opérateur "<CFOIS>" dont le facteur de droite vaudrait "1" sera remplacé par le facteur de gauche;
- l'opérateur "<CEXP>" dont l'élément de droite serait nul sera remplacé par un "<RAT>" de valeur "1";

- l'opérateur "<CEXP>" dont l'élément de droite vaudrait "1" sera remplacé par son opérande de gauche;

- l'opérateur "<CEXP>" dont l'élément de droite vaudrait "-1" sera remplacé par l'inverse (fonction INVERSER) de son opérande de gauche;
- un identificateur de fonction dont le successeur ("SUITE") serait l'identificateur de fonction inverse qui aurait ses attributs "COFACTEUR" et "PUISS" égaux à l'unité, sera remplacé par le successeur de son successeur.
- un élément de type "<VARIABLE>" ou "<ID_FONC>" dont l'attribut "COFACTEUR" serait nul sera remplacé par "Ø"
- un élément de type "<VARIABLE>" ou "<ID_FONC>" dont l'attribut "PUISS" vaudrait "Ø" sera remplacé par un "<RAT>" qui prendra la valeur de son "COFACTEUR"

13) function derivee(poirier:arbre):arbre; (listing p.47-52)

Cette fonction crée et retourne un arbre correspondant à la dérivée de l'expression représentée par l'arbre "poirier", passé en entrée.

Elle implémente l'algorithme de dérivation développé au chapitre précédent (voir p.38-42).
A chaque type d'arbre correspond une règle de dérivation. Selon la règle, elle fera ou non usage de la récursivité.

14) Programme principal (listing p.47-52)

Ce programme n'a d'autre prétention que de mettre en oeuvre les différentes procédures et fonctions pour tester leur fonctionnement.
Il permet néanmoins de se faire une petite idée des possibilités des outils que nous avons développés.

IV. CONCLUSIONS

En guise de conclusions de ce présent chapitre, nous joignons en annexe quelques exemples de simplifications et de dérivations exécutées par nos procédures.

Nous avons également joint au présent document, une disquette contenant les programmes sources ainsi qu'une version exécutable permettant au lecteur de voir par lui même l'état d'avancement de notre implémentation.

CONCLUSIONS

Rappelons au lecteur à la recherche de bibliographie sur le présent sujet, qu'il ne nous a malheureusement, pas été possible d'obtenir une indication, quelle qu'elle soit, sur la simplification et la dérivation des fonctions d'une variable réelle. Toutes les notions utilisées proviennent de l'application, des bases que nous avons reçues durant nos études, à l'intuition que nous avons eu de la solution du problème.

Cette étude, constitue donc une approche personnelle que nous nous sommes faite du sujet. Elle n'est sans doute pas nouvelle et ne constitue sans doute pas un apport théorique, mais elle constitue un pas dans l'élaboration de projets plus vaste.
Ce présent travail se veut avant tout être un outil facilement accessible pour tous développements ultérieurs de logiciels dans le domaine des expressions algébriques et tout particulièrement dans le cadre de la poursuite de l'implémentation d'un didacticiel sur la dérivation des fonctions.
Nous espérons donc que la perception que nous nous faite de la représentation et de la manipulation des expressions algébriques suscitera des émules et que nos outils trouveront une destination utile.

Terminons enfin en disant que, la réalisation de ce travail nous à permis et procuré le plaisir de nous engager dans la recherche d'une solution à un problème complexe et, nous à donner l'occasion de mettre en oeuvre les connaissances acquises durant nos études.

BIBLIOGRAPHIE

Borland International (1988) TURBO PASCAL Reference Guide, Version 5.Ø

Borland International (1988) TURBO PASCAL User's Guide, Version 5.Ø

GIECK K., FORMULAIRE technique, 6ème édition française 1979, Gieck-Verlag, heibronn, R.F.A.

BARBANSON W., Calcul différentiel et intégral, Editions des Etudiant de la FACULTE POLYTECHNIQUE de MONS

LEROY H., DE MARNEFFE P.A., Cours de COMPILATION, MATIERES APPROFONDIES, exposé en 2ème Lic. et M. Info., F.U.N.D.P Namur

ANNEXES

I. EXEMPLES DE DEMONSTRATION DU PROGRAMME

II. LISTING

III. DISQUETTE

Contenu:

- DERIVEE.PAS
- DERIVEE.EXE
- DERIVEE1.PAS
- DERIVEE1.TPU

```
                          DERIVEE
 Entrer ØØØ pour terminer  !
 Expression       = X^(-X)
 Simplification   = X^(-X)
 Derivee          = X^(-X)*(-LN[X]-1)

 Expression       = COS(2X)^3
 Simplification   = COS[2X]^3
 Derivee          = -6COS[2X]^2*SIN[2X]

 Expression       = (X+2*X^2)*(X^2+X+2)*X
 Simplification   = (2X^2+X)*(X^2+X+2)*X
 Derivee          = ((4X+1)*(X^2+X+2)+(2X+1)*(2X^2+X))*X+(X^2+X+2)*(2X^
2+X)

 Expression       = (X+COS(X))/(SIN(X)/TG(X)+X)
 Simplification   = 1
 Derivee          = Ø

 Expression       = EXP(COS(ASIN(2*LN(X/2))))
 Simplification   = EXP[COS[ASIN[2LN[(1/2)X]]]]
 Derivee          = (-2LN[(1/2)X]^2+1)^(-1/2)*-4EXP[COS[ASIN[2LN[(1/2)X
]]]]*LN[(1/2)X]/X

 Expression       = EXP(COS(ACOS(2*LN(X/2))))
 Simplification   = (1/2)X^2
 Derivee          = X

 Expression       = COS(X)^X
 Simplification   = COS[X]^(X)
 Derivee          = COS[X]^(X)*(-COS[X]^-1*SIN[X]*X+LN[COS[X]])

 Expression       = EXP(2X)
 Simplification   = EXP[2X]
 Derivee          = 2EXP[2X]

 Expression       = LN(COS(X))
 Simplification   = LN[COS[X]]
 Derivee          = -COS[X]^-1*SIN[X]

 Expression       = (X+2
 ERREUR : Niveau parenthèses incorrect !
 Expression       = 1/5 X^2*X^-3
 Simplification   = (1/5)X^-1
 Derivee          = (-1/5)X^-2

 Expression       = COS(1/3X)^(-1/2)
 Simplification   = COS[(1/3)X]^(-1/2)
 Derivee          = (1/6)COS[(1/3)X]^(-3/2)*SIN[(1/3)X]

 Expression       = COS(X)*EXP(-3LN(X^3))
 Simplification   = COS[X]/X^9
 Derivee          = -SIN[X]/X^9-9COS[X]/X^1Ø

 Expression       = 1/7COS(3/2X)
 Simplification   = (1/7)COS[(3/2)X]
 Derivee          = (-3/14)SIN[(3/2)X]

 Expression       = 2/3TG(X)^Ø
 Simplification   = (2/3)
 Derivee          = Ø
```

```
UNIT derivee1;

INTERFACE

TYPE

fonction = (FRIEN,FSIN,FCOS,FTG,FCOTG,FASIN,FACOS,FATG,FACOTG,FLN,FEXP);

rationnel=RECORD     {***  Denomi=-1 ==> real ***}
                nomi:real;
                denomi:real;
          END;
prationnel= ^rationnel;
pbyte= ^byte;
arbre_symbol = (express,terme,facteur,cvaleur,fonc,rat,variable,id_fonc,
                cplus,cmoins,cfois,cdiv,cexp,gpar,dpar,vide);
arbre = ^type_arbre;
type_arbre=RECORD
                 nom1:arbre_symbol;
                 suivant:arbre;
                 CASE nom2:arbre_symbol OF
                      cplus,cmoins,cfois,cdiv,cexp:(un,deux:arbre);
                      variable,id_fonc:(cofacteur,puiss:prationnel;
                                nomf:fonction;suite:arbre);
                      rat:(valeur:prationnel);
                      vide,gpar,dpar:();
                 END;

arbre2 = ^type_arbre2;
type_arbre2 = RECORD
                  boom:arbre;
                  message:STRING;
            END;


type_sre = (s,r,e,u);

CONST

nb_fonc=19;
list_fonc:ARRAY [1..nb_fonc]OF STRING=('SIN','COS','TG','TAN','COTG','ASIN',
                   'ARCSIN','ACOS','ARCOS','ARCTAN','ARCTG','ATG','ATAN','ACOTG'
                   'ARCOTG','ACOTAN','ARCOTAN','LN','EXP');
list_nr_fonc:ARRAY [1..nb_fonc] OF integer=(1,2,3,3,4,5,5,6,6,7,7,7,7,8,8,8,8,9,

sre:ARRAY [express..vide,rat..vide] OF type_sre
          = ((e,e,e,s,s,e,e,e,e,s,r),
             (e,e,e,r,r,s,s,e,e,r,r),
             (e,e,e,r,r,r,r,s,e,r,r),
             (e,e,e,r,r,r,r,r,e,r,r),
             (e,e,e,r,r,r,r,r,e,r,r),
             (e,u,u,r,r,r,r,r,e,r,r),
             (u,u,u,r,r,r,r,r,e,r,r),
             (s,s,s,e,s,e,e,e,s,e,e),
             (s,s,s,e,s,e,e,e,s,e,e),
             (s,s,s,e,s,e,e,e,s,e,e),
             (s,s,s,e,s,e,e,e,s,e,e),
             (s,s,s,e,s,e,e,e,s,e,e),
             (s,s,s,e,s,e,e,e,s,e,e),
             (s,s,s,e,s,e,e,e,s,e,e),
```

```
            (e,e,e,r,r,r,r,r,e,r,r),
            (s,s,s,e,s,e,e,e,s,e,e));

VAR
pun:pbyte;
strerror:STRING;
erreur:boolean;

FUNCTION asin(x:real):real;
FUNCTION acos(x:real):real;
FUNCTION tg(x:real):real;
FUNCTION cotg(x:real):real;
FUNCTION acotg(x:real):real;
FUNCTION pgcd (un,deux:real):real;
PROCEDURE simplifie (un:prationnel);
PROCEDURE plus(un,deux,trois:prationnel);
PROCEDURE moins(un,deux,trois:prationnel);
PROCEDURE fois(un,deux,trois:prationnel);
PROCEDURE divis(un,deux,trois:prationnel);
PROCEDURE puissance (un,deux,trois:prationnel);
PROCEDURE inverser_rat(entree:prationnel);
PROCEDURE effacer_arbre(entree:arbre);
FUNCTION var_to_rat(entree:arbre):arbre;
FUNCTION identique(un,deux:arbre):boolean;
FUNCTION lire_arbre(entree:arbre;priorite:pbyte):STRING;
FUNCTION calcul_arbre(pommier:arbre;x:real):real;
PROCEDURE do_arbre(donnee:arbre2);

IMPLEMENTATION

FUNCTION asin(x:real):real;  { Retourne le SINUS de l'argument }
{************************}
VAR
a,b,c,y:real;
BEGIN
IF abs(x)>1 THEN
BEGIN
     asin:=999;
     strerror:='Hors du domaine de ASIN';
     erreur:=true;
END ELSE
BEGIN
b:=pi/2;
a:=-b;
c:=0;
y:=0;
WHILE (abs(y-x)>0.0000000001) DO
BEGIN
     IF x>y THEN
          a:=c
     ELSE b:=c;
     c:=(a+b)/2;
     y:=sin(c);
END;
asin:=c;
END;
END;

FUNCTION acos(x:real):real;  { Retourne le COSINUS de l'argument }
{************************}
```

```
VAR
a,b,c,y:real;
BEGIN
IF abs(x)>1 THEN
BEGIN
     acos:=999;
     strerror:='Hors du domaine de ACOS';
     erreur:=true;
END ELSE
BEGIN
b:=pi;
a:=0;
c:=pi/2;
y:=0;
WHILE (abs(y-x)>0.0000000001) DO
BEGIN
     IF x<y THEN
          a:=c
     ELSE b:=c;
     c:=(a+b)/2;
     y:=cos(c);
END;
acos:=c;
END;
END;

FUNCTION tg(x:real):real;  { Retourne la TANGENTE de l'argument }
{************************}
BEGIN
     tg:=sin(x)/cos(x);
END;

FUNCTION cotg(x:real):real;  { Retourne la COTANGENTE de l'argument }
{************************}
BEGIN
     cotg:=cos(x)/sin(x);
END;

FUNCTION acotg(x:real):real;  { Retourne l' ARCOTANGENTE de l'argument }
{************************}
BEGIN
     acotg:=pi/2-arctan(x);
END;

FUNCTION pgcd (un,deux:real):real;  { Retourne le PGCD des 2 arguments }
{*******************************}
CONST
premier : ARRAY [1..42] OF real=(2,3,5,7,11,13,17,19,23,29,31,37,
            41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,109,
            113,127,131,137,139,149,151,157,163,167,173,179,181);
VAR
commun:real;
i:integer;
racine:real;
BEGIN
   IF un=0 THEN IF deux<>0 THEN pgcd:=deux ELSE pgcd:=1
   ELSE IF deux=0 THEN pgcd:=un ELSE
   BEGIN
     pgcd:=1;
     IF un>deux THEN
```

```
     BEGIN
          un:=un+deux;
          deux:=un-deux;
          un:=un-deux;
     END;
     IF frac(deux / un)=0 THEN pgcd:=un
     ELSE BEGIN
          commun:=1;
          i:=1;
          racine:=int(sqrt(deux));
          WHILE (premier[i]<=racine) AND (premier[i]<=un) DO
               IF (frac(un / premier[i])=0) AND (frac(deux / premier[i])=0) THEN
               BEGIN
                    un:=round(un/premier[i]);
                    deux:=round(deux/premier[i]);
                    commun:=round(commun*premier[i]);
                    racine:=int(sqrt(deux));
               END
               ELSE i:=i+1;
          pgcd:=commun;
          END;
   END;
END;

PROCEDURE simplifie (un:prationnel);  { Simplifie un RATIONNEL ou un REEL}
{*********************************}
VAR
commun:real;
BEGIN
   IF (un^.denomi=0) THEN
   BEGIN
        strerror:=' Denominateur = 0';
        erreur:=true;
   END ELSE
   BEGIN
     IF (un^.denomi>0) THEN
     BEGIN
          commun:=pgcd(abs(un^.nomi),abs(un^.denomi));
          un^.nomi:=round(un^.nomi/commun);
          un^.denomi:=round(un^.denomi/commun);
     END ELSE
     BEGIN
          IF frac (un^.nomi)=0 THEN un^.denomi:=-1*un^.denomi;
     END;
  END;
END;

PROCEDURE plus(un,deux,trois:prationnel); { trois := un + deux }
{************************************}
VAR
   commun:real;
   aux:rationnel;
   a,b,c,d:real;
BEGIN
   IF (un^.denomi=0) OR (deux^.denomi=0) THEN
   BEGIN
        strerror:=' Denominateur = 0';
        erreur:=true;
   END ELSE
   BEGIN
```

```
      IF (un^.denomi>0) AND (deux^.denomi>0) THEN
      BEGIN
           commun:=pgcd(un^.denomi,deux^.denomi);
           a:=round(un^.denomi/commun);
           b:=round(deux^.denomi/commun);
           aux.denomi:=round(un^.denomi*b);
           aux.nomi:=round(un^.nomi*b+deux^.nomi*a);
           simplifie(addr(aux));
           trois^:=aux;
      END ELSE
      BEGIN
           IF un^.denomi=-1 THEN c:=un^.nomi
           ELSE c:=un^.nomi/un^.denomi;
           IF deux^.denomi=-1 THEN d:=deux^.nomi
           ELSE d:=deux^.nomi/deux^.denomi;
           aux.denomi:=-1;
           aux.nomi:=c+d;
           IF aux.nomi= int(aux.nomi) THEN aux.denomi:=1;
           trois^:=aux;
      END;
    END;
END;

PROCEDURE moins(un,deux,trois:prationnel);  { trois := un - deux }
{*************************************}
VAR
   commun:real;
   aux:rationnel;
   a,b,c,d:real;
BEGIN
   IF (un^.denomi=0) OR (deux^.denomi=0) THEN
   BEGIN
        strerror:=' Denominateur = 0';
        erreur:=true;
   END ELSE
   BEGIN
     IF (un^.denomi>0) AND (deux^.denomi>0) THEN
     BEGIN
           commun:=pgcd(un^.denomi,deux^.denomi);
           a:=round(un^.denomi/commun);
           b:=round(deux^.denomi/commun);
           aux.denomi:=round(un^.denomi*b);
           aux.nomi:=round(un^.nomi*b-deux^.nomi*a);
           simplifie(addr(aux));
           trois^:=aux;
     END ELSE
     BEGIN
           IF un^.denomi=-1 THEN c:=un^.nomi
           ELSE c:=un^.nomi/un^.denomi;
           IF deux^.denomi=-1 THEN d:=deux^.nomi
           ELSE d:=deux^.nomi/deux^.denomi;
           aux.denomi:=-1;
           aux.nomi:=a-b;
           IF aux.nomi= int(aux.nomi) THEN aux.denomi:=1;
           trois^:=aux;
     END;
   END;
END;

PROCEDURE fois(un,deux,trois:prationnel);  { trois := un * deux }
```

```
{*************************************}
VAR
   aux:rationnel;
   a,b:real;
BEGIN
   IF (unˆ.denomi=Ø) OR (deuxˆ.denomi=Ø) THEN
   BEGIN
        strerror:=' Denominateur = Ø';
        erreur:=true;
   END ELSE
   BEGIN
     IF (unˆ.denomi>Ø) AND (deuxˆ.denomi>Ø) THEN
     BEGIN
          aux.denomi:=round(unˆ.denomi*deuxˆ.denomi);
          aux.nomi:=round(unˆ.nomi*deuxˆ.nomi);
          simplifie(addr(aux));
          troisˆ:=aux;
     END ELSE
     BEGIN
          IF unˆ.denomi=-1 THEN a:=unˆ.nomi
          ELSE a:=unˆ.nomi/unˆ.denomi;
          IF deuxˆ.denomi=-1 THEN b:=deuxˆ.nomi
          ELSE b:=deuxˆ.nomi/deuxˆ.denomi;
          aux.denomi:=-1;
          aux.nomi:=a*b;
          IF aux.nomi= int(aux.nomi) THEN aux.denomi:=1;
          troisˆ:=aux;
     END;
  END;
END;

PROCEDURE divis(un,deux,trois:prationnel);  { trois := un / deux }
{*************************************}
VAR
   aux:rationnel;
   a,b:real;
BEGIN
   IF (unˆ.denomi=Ø) OR (deuxˆ.denomi=Ø) THEN
   BEGIN
        strerror:=' Denominateur = Ø';
        erreur:=true;
   END ELSE IF (deuxˆ.nomi=Ø) THEN
   BEGIN
        strerror:=' Division par Ø';
        erreur:=true;
   END ELSE
   BEGIN
     IF (unˆ.denomi>Ø) AND (deuxˆ.denomi>Ø) THEN
     BEGIN
          aux.denomi:=round(unˆ.denomi*deuxˆ.nomi);
          aux.nomi:=round(unˆ.nomi*deuxˆ.denomi);
          simplifie(addr(aux));
          troisˆ:=aux;
     END ELSE
     BEGIN
          IF unˆ.denomi=-1 THEN a:=unˆ.nomi
          ELSE a:=unˆ.nomi/unˆ.denomi;
          IF deuxˆ.denomi=-1 THEN b:=deuxˆ.nomi
          ELSE b:=deuxˆ.nomi/deuxˆ.denomi;
          aux.denomi:=-1;
```

```
          aux.nomi:=a/b;
          IF aux.nomi= int(aux.nomi) THEN aux.denomi:=1;
          trois^:=aux;
     END;
  END;
END;

PROCEDURE puissance (un,deux,trois:prationnel);  { trois := un ^ deux }
{*******************************************}
VAR
   commun:real;
   aux:rationnel;
   a,b,c,d:real;
BEGIN
   IF (un^.denomi=0) OR (deux^.denomi=0) THEN
   BEGIN
        strerror:=' Denominateur = 0';
        erreur:=true;
   END ELSE
   BEGIN
     simplifie(deux);
     IF (un^.denomi>0) AND (deux^.denomi=1) THEN
     BEGIN
          aux.denomi:=round(exp(ln(un^.denomi)*deux^.nomi));
          aux.nomi:=  round(exp(ln(un^.nomi)*deux^.nomi));
          simplifie(addr(aux));
          trois^:=aux;
     END ELSE
     BEGIN
          IF un^.denomi=-1 THEN a:=un^.nomi
          ELSE a:=un^.nomi/un^.denomi;
          IF deux^.denomi=-1 THEN b:=deux^.nomi
          ELSE b:=deux^.nomi/deux^.denomi;
          aux.denomi:=-1;
          aux.nomi:=exp(ln(a)*b);
          simplifie(addr(aux));
          trois^:=aux;
     END;
  END;
END;

PROCEDURE inverser_rat(entree:prationnel);  { entree := entree^-1 }
{*************************************}
VAR aux:real;
BEGIN
     IF (entree^.denomi>0) THEN
     BEGIN
          IF entree^.nomi<>0 THEN
          BEGIN
               aux:=entree^.nomi;
               entree^.nomi:=entree^.denomi;
               entree^.denomi:=aux;
               IF entree^.denomi<0 THEN
               BEGIN
                    entree^.nomi:=-1*entree^.nomi;
                    entree^.denomi:=-1*entree^.denomi;
               END;
          END ELSE
          BEGIN
              strerror:=' Denominateur = 0';
```

```
            erreur:=true;
        END;
    END ELSE entree^.nomi:=-1/entree^.nomi;
END;

FUNCTION var_to_rat(entree:arbre):arbre; {  Cofacteur*XXX^Ø ==> Cofacteur=rat}
{************************************}
VAR aux:arbre;
BEGIN
     IF (entree^.nom2 IN [variable..id_fonc])
        AND (entree^.puiss^.nomi=Ø) THEN
     BEGIN
          new(aux);
          aux^.nom1:=entree^.nom1;
          aux^.nom2:=rat;
          new(aux^.valeur);
          aux^.valeur^:=entree^.cofacteur^;
          aux^.suivant:=NIL;
          effacer_arbre (entree);
          var_to_rat:=aux;
     END ELSE var_to_rat:=entree;
END;

FUNCTION identique(un,deux:arbre):boolean; { TRUE si arbreA=arbreB}
{************************************}
FUNCTION egal(un,deux:prationnel):boolean; { Teste l'egalite de 2 rationnel }
{************************************}
BEGIN
     IF (un^.nomi=deux^.nomi) AND
        (un^.denomi=deux^.denomi)
     THEN egal:=true ELSE egal:=false;
END;

BEGIN  {***************** Debut fonction Identique ***************}
     IF un^.nom2<>deux^.nom2 THEN identique:=false ELSE
     BEGIN
          identique:=false;
          CASE un^.nom2 OF
               cplus,cfois:IF (identique(un^.un,deux^.un)
                           AND identique(un^.deux,deux^.deux))
                           OR (identique(un^.un,deux^.deux)
                           AND identique(un^.deux,deux^.un))
                         THEN identique:=true;
               cmoins,cdiv,cexp:IF (identique(un^.un,deux^.un))
                           AND (identique(un^.deux,deux^.deux))
                         THEN identique:=true;
               rat:IF egal(un^.valeur,deux^.valeur)
                         THEN identique:=true;
               variable:IF egal(un^.cofacteur,deux^.cofacteur) AND
                           egal(un^.puiss,deux^.puiss)
                         THEN identique:=true;
               id_fonc:IF (un^.nomf=deux^.nomf) AND
                          egal(un^.cofacteur,deux^.cofacteur) AND
                          egal(un^.puiss,deux^.puiss) AND
                          identique(un^.suite,deux^.suite)
                         THEN identique:=true;
          END;
     END;
END; {************************ Fin fonction Identique *************}
```

```
PROCEDURE effacer_arbre(entree:arbre); { Efface du HRAP tous les pointeurs }
{*********************************} { constituants l'arbre Entree       }
BEGIN


     CASE entree^.nom2 OF
          rat: dispose(entree^.valeur);
          variable:
                  BEGIN
                       dispose(entree^.cofacteur);
                       dispose(entree^.puiss);
                  END;
          id_fonc:BEGIN
                       effacer_arbre (entree^.suite);
                       dispose(entree^.cofacteur);
                       dispose(entree^.puiss);
                  END;
          cplus..cexp:BEGIN
                           effacer_arbre (entree^.un);
                           effacer_arbre (entree^.deux);
                      END;
          END;
          IF entree<> NIL THEN dispose(entree);
END;


FUNCTION lire_arbre(entree:arbre;priorite:pbyte):STRING;
{*****************************************************}

     {  Fonction construisant recursivement la representation externe }
     {  de l'expression algebrique, dont la representation interne est }
     { un arbre binaire formé de pointeurs }

VAR
sortie,sa,sb,sc,sd,se,sf,sg,sh:STRING;
prio,copyprio:byte;
gauche:boolean;
rat_aux:prationnel;

CONST
     signe :ARRAY [cplus..cexp] OF STRING=('+','-','*','/','^');
     isigne:ARRAY [cplus..cexp] OF STRING=('-','+','/','*','^');
     rep_fonc:ARRAY [FSIN..FEXP] OF STRING=('SIN','COS','TG','CTG',
                'ASIN','ACOS','ATG','ACOTG','LN','EXP');
FUNCTION rat_to_str(a:prationnel):STRING;
{***************************************}
VAR

   lnomi,ldenomi:integer;
   sa,sb,sc,sd,se:STRING;
BEGIN
     sa:='';
     sb:='';
     sc:='';
     sd:='';
     se:='';
     IF a^.nomi=0 THEN
     BEGIN
          sb:='0';
     END ELSE
```

```
     IF a^.denomi=0 THEN
     BEGIN
          strerror:='denominateur=0';
          erreur:=true;
          rat_to_str:='';
          exit
     END ELSE

     IF a^.denomi > 0 THEN
     BEGIN
          sc:='/';
          lnomi:=round(int(ln(abs(a^.nomi))/ln(10))+1);
          str(a^.nomi :lnomi :0,sb);
           ldenomi:=round(int(ln(abs(a^.denomi))/ln(10))+1);
          str(a^.denomi :ldenomi :0,sd);
          IF (sd='1') THEN
          BEGIN
               sa:='';
               sc:='';
               sd:='';
               se:='';
          END ELSE IF (sd<>'') THEN
          BEGIN
               sa:='(';
               se:=')';
          END;
     END ELSE
     BEGIN
          str(a^.nomi,sb);
          sa:='';
          sc:='';
          sd:='';
          se:='';
     END;
     rat_to_str:=sa+sb+sc+sd+se;
END;

BEGIN   {***************   debut fonction lire_arbre    ************}
     sortie:='';
     sa:='';
     sb:='';
     sc:='';
     sd:='';
     se:='';
     sf:='';
     sg:='';
     sh:='';
     IF priorite^>4 THEN
     BEGIN
          gauche:=false;
          priorite^:=priorite^-4;
     END ELSE gauche:=true;
     CASE entree^.nom2 OF
          cplus..cexp:
            BEGIN
                 IF (entree^.nom2 IN[cplus..cmoins]) THEN prio:=1;
                 IF (entree^.nom2 IN[cfois..cdiv]) THEN prio:=2;
                 IF (entree^.nom2=cexp) THEN prio:=3;
                 IF (priorite^>prio) THEN
                 BEGIN
```

```
            sa:='(';
            sf:=')';
        END ELSE
        BEGIN
            sa:='';
            sf:='';
        END;
        IF ((prio=3)AND(entree^.deux^.nom2<>rat)) THEN
        BEGIN
            sg:='(';
            sh:=')';
        END ELSE
        BEGIN
            sg:='';
            sh:='';
        END;

        copyprio:=prio;
        sb:=lire_arbre(entree^.un,addr(copyprio));
        copyprio:=prio + 4;
        sd:=lire_arbre(entree^.deux,addr(copyprio));
        IF (copy(sd,1,1)='(' ) THEN
        BEGIN
            sg:='';
            sh:='';
        END;
        IF (copyprio>(prio+4)) THEN
        BEGIN
            sc:=isigne[entree^.nom2];
        END
        ELSE sc:=signe[entree^.nom2];
        sortie:=sa+sb+sc+sg+sd+sh+sf;
  END;
variable..id_fonc:
     BEGIN
        IF (NOT gauche) AND (priorite^=1)
           AND (entree^.cofacteur^.nomi<0) THEN
        BEGIN
            priorite^:=11;
            sa:=copy(rat_to_str(entree^.cofacteur),2,69);
            sc:=rat_to_str(entree^.puiss);
        END ELSE IF (NOT gauche) AND (priorite^=2)
           AND (entree^.puiss^.nomi<0) THEN
        BEGIN
            priorite^:=12;
            new(rat_aux);
            rat_aux^:=entree^.cofacteur^;
            inverser_rat(rat_aux);
            sa:=rat_to_str(rat_aux);
            dispose(rat_aux);
            sc:=copy(rat_to_str(entree^.puiss),2,69);
        END ELSE
        BEGIN
            sa:=rat_to_str(entree^.cofacteur);
            sc:=rat_to_str(entree^.puiss);
        END;
        IF (sa<>'0') THEN
        BEGIN
            IF (sc<>'0') THEN
            BEGIN
```

```
                    IF sa='1' THEN sa:='';
                    IF sa='-1' THEN sa:='-';
                    IF sc='1' THEN sc:='' ELSE sc:='^'+sc;
                    IF entree^.nom2=variable THEN
                    BEGIN
                         sb:='X';
                         sd:=''
                    END ELSE
                    BEGIN
                         sb:=rep_fonc[entree^.nomf];
                         sd:='['+lire_arbre(entree^.suite,pun)+']';
                    END;
               END ELSE
               BEGIN
                    sb:='';
                    sd:='';
                    sc:='';
               END;
          END ELSE
          BEGIN
               sb:='';
               sc:='';
               sd:='';
          END;
          sortie:=sa+sb+sd+sc;
      END;
    rat:
      BEGIN
          IF (NOT gauche) AND (priorite^=1)
             AND (entree^.valeur^.nomi<0) THEN
          BEGIN
                priorite^:=11;
             sortie:=copy(rat_to_str(entree^.valeur),2,69);
          END ELSE IF (NOT gauche) AND (priorite^=2)
             AND (abs(entree^.valeur^.nomi)=1)
             AND (entree^.valeur^.denomi>0) THEN
          BEGIN
               priorite^:=12;
               new(rat_aux);
               rat_aux^:=entree^.valeur^;
               inverser_rat(rat_aux);
               sortie:=rat_to_str(rat_aux);
               dispose(rat_aux);
          END ELSE sortie:=rat_to_str(entree^.valeur);
      END;
    ELSE
      BEGIN
          sortie :='@';
      END;
  END;
IF (priorite^=1) AND (copy(sortie,1,1)='-') AND (NOT gauche) THEN
BEGIN
  sortie:=copy(sortie,2,250);
  priorite^:=11;
END;
lire_arbre:=sortie;
END;
{************************ fin fonction lire_arbre **********************}
```

```
FUNCTION calcul_arbre(pommier:arbre;x:real):real;
{*********************************************}

     { fonction qui evalue l'expression pour une valeur particuliere de X }

VAR a,b,aux:real;
f:fonction;


BEGIN
     CASE pommier^.nom2 OF
     variable:BEGIN
                    IF pommier^.cofacteur^.denomi<>-1
                    THEN a:=pommier^.cofacteur^.nomi/pommier^.cofacteur^.denomi
                    ELSE a:=pommier^.cofacteur^.nomi;
                    IF pommier^.puiss^.denomi<>-1
                    THEN b:=pommier^.puiss^.nomi/pommier^.puiss^.denomi
                    ELSE b:=pommier^.puiss^.nomi;
                    aux:=a*(exp(ln(x)*b));
              END;
     rat:     BEGIN
                    IF pommier^.valeur^.denomi<>-1
                    THEN aux:=pommier^.valeur^.nomi/pommier^.valeur^.denomi
                    ELSE aux:=pommier^.valeur^.nomi;
              END;
     cplus:   BEGIN
                    a:=calcul_arbre(pommier^.un,x);
                    b:=calcul_arbre(pommier^.deux,x);
                    aux:=a+b;
              END;
     cmoins:  BEGIN
                    a:=calcul_arbre(pommier^.un,x);
                    b:=calcul_arbre(pommier^.deux,x);
                    aux:=a-b;
              END;
     cfois:   BEGIN
                    a:=calcul_arbre(pommier^.un,x);
                    b:=calcul_arbre(pommier^.deux,x);
                    aux:=a*b;
              END;
     cdiv:    BEGIN
                    a:=calcul_arbre(pommier^.un,x);
                    b:=calcul_arbre(pommier^.deux,x);
                    IF b=0 THEN aux:=999.999
                    ELSE aux:=a/b;
              END;
     cexp:    BEGIN
                    a:=calcul_arbre(pommier^.un,x);
                    b:=calcul_arbre(pommier^.deux,x);
                    IF abs(b-1)<0.000000001 THEN aux:=1
                    ELSE IF a>0 THEN aux:=exp(ln(a)*b)
                    ELSE aux:=-exp(ln(a)*b) { ?? danger !!}
              END;
     id_fonc: BEGIN
                    a:=calcul_arbre(pommier^.suite,x);
                    f:=pommier^.nomf;
                    IF f= FSIN  THEN aux:=SIN(a)
                    ELSE IF f= FCOS   THEN aux:=COS(a)
                    ELSE IF f= FTG    THEN aux:=TG(a)
                    ELSE IF f= FCOTG  THEN aux:=COTG(a)
```

```
                    ELSE IF f= FASIN  THEN aux:=ASIN(a)
                    ELSE IF f= FACOS  THEN aux:=ACOS(a)
                    ELSE IF f= FATG   THEN aux:=ARCTAN(a)
                    ELSE IF f= FACOTG THEN aux:=ACOTG(a)
                    ELSE IF f= FLN    THEN aux:=LN(a)
                    ELSE IF f= FEXP   THEN aux:=EXP(a);
               END;
     ELSE aux:=69;
     END;
     calcul_arbre:=aux;
END;

PROCEDURE do_arbre(donnee:arbre2);
{*********************************}

    { procedure principale de mise sous forme d'arbre binaire,  }
    { d'une expression entree sous forme de caracteres          }

VAR
{***}
i,j,k,l,index:integer;
nomf,valeur,sortie,entree,message_erreur:STRING;
valeure:real;
tampon:char;
failure,fin_entree,ok:boolean;
mot:arbre_symbol;
stack,input,nouveau1,nouveau2:arbre;

FUNCTION litchar:char;
{********************}
BEGIN
     litchar:=entree[index];
     index:=index+1;
END;


PROCEDURE shift;  { Extrait le symbole algebrique terminal du dessus de la }
{**************}  { pile SHIFT vers le dessus de la pile STACK et recharge }
                  { la pile SHIFT du symbole terminal suivant               }
CONST
     tchiffre :SET OF char=['0'..'9'];
     tchiffre2:SET OF char=['0'..'9',',','.'];
     tchiffre3:SET OF char=['0'..'9','_',',',','.'];
     tlettre  :SET OF char=['a'..'z','A'..'Z'];
VAR nouveau:arbre;
tmpval,tmpcof,tmppuis:prationnel;
tmpnomf:fonction;
tmpnom:arbre_symbol;

PROCEDURE cas_variable(chaine:STRING);
{-----------------------------------}


BEGIN  {---------------------  debut de la procedure cas_variable ----}
    IF chaine='PI' THEN
    BEGIN
        new(tmpval);
          tmpnom:=rat;
        tmpval^.nomi:=pi;
        tmpval^.denomi:=-1;
```

```
    END
    ELSE IF chaine='E' THEN
           BEGIN
                 new(tmpval);
          tmpnom:=rat;
                    tmpval^.nomi:=exp(1);
                    tmpval^.denomi:=-1;
          END ELSE
           BEGIN
              new(tmppuis);
              new(tmpcof);
                tmpnom:=variable;
              tmppuis^.nomi:=1;
              tmppuis^.denomi:=1;
              tmpcof^.nomi:=1;
              tmpcof^.denomi:=1;
          END;
END;

PROCEDURE cas_nombre;
{------------------}

VAR ok,dejapoint:Boolean;
ce:integer;
strvaleur:STRING;
valeur:real;

BEGIN  {---------------------- debut procedure cas_nombre -----------}
       ok:=true;
       dejapoint:=false;
       strvaleur:='';
       IF (tampon='.') OR (tampon=',') THEN
       BEGIN
            strvaleur:='0';
            dejapoint:=true;
       END;
       IF tampon='_' THEN tampon:='-';
       IF tampon=',' THEN tampon:='.';
       strvaleur:=strvaleur+tampon;
       tampon:=litchar;
       WHILE ((tampon IN tchiffre2) AND ok ) DO
             BEGIN
                  IF (  tampon ='.') OR (tampon = ',') THEN
                  BEGIN
                              IF tampon=',' THEN tampon:='.';
                              IF dejapoint THEN
                              BEGIN
                                      erreur:=true;
                                      strerror:='Double virgule : "'+strvaleur+'
                                      ok:=false;
                              END ELSE
                              BEGIN
                                      dejapoint:=true;
                                      strvaleur := strvaleur + tampon;
                                      tampon:=litchar;
                              END;
                   END ELSE IF (tampon IN tchiffre) THEN
                            BEGIN
                                     strvaleur := strvaleur + tampon;
                                     tampon:=litchar;
```

```
                              END ELSE ok:=false;
            END;
     tmpnom := rat;
     new(tmpval);
     val(strvaleur,valeur,ce);
     IF valeur=int(valeur) THEN
                    tmpvalˆ.denomi:=1 ELSE tmpvalˆ.denomi:=-1;
     tmpvalˆ.nomi:=valeur;
END;  {-------------------- fin procedure cas nombre ----------------}

PROCEDURE   cas_mot;
{-----------------}
VAR
   Uvaleur:STRING;
   nrf:integer;

FUNCTION nr_fonc(Uvaleur:STRING):integer; { Retourne le numero d'ordre de }
{************************************} { l'eventuelle fonction corres- }
                                        { pondant a la chaine de caract-}
                                        { entree                        }
VAR i:integer;
nrf:integer;
BEGIN
     i:=1;
     nr_fonc:=0;
     WHILE i<(nb_fonc+1) DO
       BEGIN
           IF Uvaleur=list_fonc[i] THEN
           BEGIN
                nr_fonc:=list_nr_fonc[i];
                i:=nb_fonc+1;
           END ELSE i:=i+1;
       END;
END;

BEGIN  {-------------------- debut procedure cas_mot ----------------}
     valeur:=tampon;
     Uvaleur:=upcase(tampon);
     tampon:=litchar;
     WHILE (tampon IN tlettre) DO
          BEGIN
               valeur:=valeur+tampon;
               Uvaleur:=Uvaleur+upcase(tampon);
               tampon:=litchar;
          END;
     nrf := nr_fonc (Uvaleur);
     IF nrf<>0 THEN
                  BEGIN
                       tmpnom:=id_fonc;
                       tmpnomf:=FSIN;
                       i:=1;
                       WHILE i < nrf DO
                       BEGIN
                            tmpnomf:=succ(tmpnomf);
                            i:=i+1;
                       END;
                    new(tmppuis);
                    new(tmpcof);
                    tmppuisˆ.nomi:=1;
                    tmppuisˆ.denomi:=1;
```

```
                    tmpcof^.nomi:=1;
                    tmpcof^.denomi:=1;
                  END ELSE cas_variable(Uvaleur);
END; {-------------------- fin procedure cas mot --------------------}

BEGIN  {------------------ debut de la procedure shift -------------}
IF input^.suivant=NIL THEN
BEGIN
     ok:=false;
     WHILE NOT ok DO
     BEGIN
          CASE tampon OF
               '#':BEGIN
                        tmpnom:=vide;
                        ok:=true;
                        fin_entree:=true;
                   END;
               '(','[':BEGIN
                        tmpnom:=gpar;
                        ok:=true;
                        tampon:=litchar;
                   END;
               ')',']':BEGIN
                        tmpnom:=dpar;
                        ok:=true;
                        tampon:=litchar;
                   END;
               '+':BEGIN
                        tmpnom:=cplus;
                        ok:=true;
                        tampon:=litchar;
                   END;
               '-':BEGIN
                        tmpnom:=cmoins;
                        ok:=true;
                        tampon:=litchar;
                   END;
               '/':BEGIN
                        tmpnom:=cdiv;
                        ok:=true;
                        tampon:=litchar;
                   END;
               '*':BEGIN
                        tampon:=litchar;
                        IF tampon='*' THEN
                        BEGIN
                             tmpnom:=cexp;
                             tampon:=litchar;
                        END
                        ELSE tmpnom:=cfois;
                        ok:=true;
                   END;
               '^':BEGIN
                        tmpnom:=cexp;
                        ok:=true;
                        tampon:=litchar;
                   END;
               ELSE
                    IF tampon IN tchiffre3 THEN
                    BEGIN
```

```
                        cas_nombre;
                        ok:=true;
                    END ELSE IF tampon IN tlettre THEN
                             BEGIN
                                  cas_mot;
                                  ok:=true;
                             END ELSE
                                 IF tampon=' 'THEN
                                 BEGIN
                                       tampon:=litchar;
                                       ok:=false;
                                 END ELSE
                                 BEGIN
                                    strerror:=copy(entree,1,index-1)+'... Caract
                                    erreur:=true;
                                    ok:=true;
                                 END;
              END;
     END;
     IF NOT erreur THEN
     BEGIN
          input^.suivant:=stack;
          stack:=input;
          New (input);
          input^.nom1:=tmpnom;
           input^.nom2:=tmpnom;
          IF (tmpnom=rat) THEN
          BEGIN
               input^.valeur:=tmpval;
          END;
          IF (tmpnom=id_fonc) THEN
          BEGIN
               input^.nomf:=tmpnomf;
               input^.cofacteur:=tmpcof;
                input^.puiss:=tmppuis;
               input^.suite:=NIL;
          END;
          IF (tmpnom=variable) THEN
           BEGIN
                   input^.cofacteur:=tmpcof;
                   input^.puiss:=tmppuis;
                input^.suite:=NIL;
                input^.nomf:=FRIEN;
           END;
          IF (tmpnom=cplus) OR
             (tmpnom=cmoins) OR
             (tmpnom=cfois) OR
             (tmpnom=cdiv) OR
             (tmpnom=cexp) THEN
          BEGIN
               input^.un:=NIL;
               input^.deux:=NIL;
          END;
          input^.suivant:=NIL;
     END;
END ELSE
BEGIN
     nouveau:=input^.suivant;
     input^.suivant:=stack;
     stack:=input;
```

```
     input:=nouveau;
END;
END; {------------------- fin de la procedure shift -----------------}

PROCEDURE stop_erreur;
{********************}
BEGIN
      message_erreur:=' ERREUR : '+strerror;
      donnee^.message:=message_erreur;
      failure:=true;
END;

PROCEDURE proc_s1; { Insere un "*" dans la pile SHIFT lors de rencontre }
{****************} { d'expression comme 3sin(2x) ==> 3*sin(2*x)         }
VAR nouveau:arbre;

BEGIN
     new(nouveau);
     IF input^.nom1=rat THEN
     BEGIN
          nouveau^.nom1:=cexp;
          nouveau^.nom2:=cexp;
     END ELSE
     BEGIN
          nouveau^.nom1:=cfois;
          nouveau^.nom2:=cfois;
     END;
     nouveau^.un:=NIL;
     nouveau^.deux:=NIL;
     nouveau^.suivant:=input;
     input:=nouveau;
END;

PROCEDURE reduce;
{***************}
VAR nouveau:arbre;

FUNCTION production:integer; { retourne le numero de la regle de reduction    }
{************************} { qui peut etre appliquer  aux elements formants }
VAR prod:integer;            { le sommet de STACK                             }
    prec:arbre_symbol;
BEGIN
CASE stack^.nom1 OF
     terme:IF (stack^.suivant^.suivant^.nom1=express) THEN
                   IF stack^.suivant^.nom1=cplus THEN prod:=1
                   ELSE IF stack^.suivant^.nom1=cmoins THEN prod:=2
                        ELSE prod:=3
           ELSE prod:=3;
     facteur:IF stack^.suivant^.suivant^.nom1=terme THEN
             CASE stack^.suivant^.nom1 OF
                       cfois:prod:=4;
                       cdiv :prod:=5;
                       ELSE prod:=6
             END ELSE prod:=6;
     cvaleur:IF (stack^.suivant^.suivant^.nom1=facteur)
                AND(stack^.suivant^.nom1=cexp) THEN prod:=7
             ELSE IF stack^.suivant^.nom1=id_fonc THEN prod:=17
                  ELSE prod:=8;
     rat:BEGIN
               IF (stack^.suivant^.nom1=cmoins) THEN
```

```
                BEGIN
                     prec:=stack^.suivant^.suivant^.nom1;
                     IF (prec=id_fonc) OR (prec=gpar) OR (prec=cplus)
                        OR (prec=cmoins) OR (prec=cfois) OR (prec=cdiv)
                        OR (prec=cexp) OR (prec=vide) THEN prod:=14
                     ELSE prod:=13;
                END ELSE prod:=13;
           END;
      fonc:BEGIN
                IF (stack^.suivant^.nom1=cmoins) THEN
                BEGIN
                     prec:=stack^.suivant^.suivant^.nom1;
                     IF (prec=id_fonc) OR (prec=gpar) OR (prec=cplus)
                        OR (prec=cmoins) OR (prec=cfois) OR (prec=cdiv)
                        OR (prec=cexp) OR (prec=vide) THEN prod:=12
                     ELSE prod:=11;
                END ELSE prod:=11;
           END;
      variable:BEGIN
                IF (stack^.suivant^.nom1=cmoins) THEN
                BEGIN
                     prec:=stack^.suivant^.suivant^.nom1;
                     IF (prec=id_fonc) OR (prec=gpar) OR (prec=cplus)
                        OR (prec=cmoins) OR (prec=cfois) OR (prec=cdiv)
                        OR (prec=cexp) OR (prec=vide) THEN prod:=16
                     ELSE prod:=15;
                END ELSE prod:=15;
           END;
      dpar:IF (stack^.suivant^.suivant^.nom1=gpar)
              AND (stack^.suivant^.nom1=express) THEN
              IF (stack^.suivant^.suivant^.suivant^.nom1=cmoins)THEN
              BEGIN
                     prec:=stack^.suivant^.suivant^.suivant^.suivant^.nom1;
                     IF (prec=id_fonc) OR (prec=gpar) OR (prec=cplus)
                        OR (prec=cmoins) OR (prec=cfois) OR (prec=cdiv)
                        OR (prec=cexp) OR (prec=vide) THEN prod:=10
                     ELSE prod:=9;
              END ELSE prod:=9
           ELSE BEGIN
                     prod:=0;
                     strerror:='Niveau de parenthèses incorrect !';
                END;
END;
production:=prod;
END; {------------------- fin de la fonction production --------------}

PROCEDURE reduce_1;{<express>:=<express><+><terme>}
{*****************}
VAR a,b,c :arbre;
BEGIN
     a:= stack;
     b:= stack^.suivant;
     c:= stack^.suivant^.suivant;
     IF (a^.nom2=rat) AND (c^.nom2=rat) THEN
     BEGIN
          plus(c^.valeur,a^.valeur,a^.valeur);
          a^.nom1:=express;
          a^.suivant:=stack^.suivant^.suivant^.suivant;
          stack:= a;
          dispose (b);
```

```
          effacer_arbre (c);
     END ELSE IF ((a^.nom2=variable) AND (c^.nom2=variable)
               AND (a^.puiss^.nomi=c^.puiss^.nomi)
               AND (a^.puiss^.denomi=c^.puiss^.denomi))
               OR
               ((a^.nom2=id_fonc) AND (c^.nom2=id_fonc)
               AND (a^.puiss^.nomi=c^.puiss^.nomi)
               AND (a^.puiss^.denomi=c^.puiss^.denomi)
               AND (a^.nomf = c^.nomf)
               AND (identique(a^.suite,c^.suite)))THEN

     BEGIN
               a^.nom1:=express;
               a^.suivant:=stack^.suivant^.suivant^.suivant;
               plus(c^.cofacteur,a^.cofacteur,a^.cofacteur);
               dispose (b);
               effacer_arbre (c);
               stack:= a;
     END ELSE
     BEGIN
          b^.un:=c;
          b^.deux:=a;
          b^.suivant:=stack^.suivant^.suivant^.suivant;
          b^.nom1:=express;
          a^.suivant:=NIL;
          c^.suivant:=NIL;
          stack:=b;
    END;
END;

PROCEDURE reduce_2;{<express>:=<express><->(terme>}
{*****************}
VAR a,b,c :arbre;
BEGIN
     a:= stack;
     b:= stack^.suivant;
     c:= stack^.suivant^.suivant;
     IF (a^.nom2=rat) AND (c^.nom2=rat) THEN
     BEGIN
          moins(c^.valeur,a^.valeur,a^.valeur);
          a^.nom1:=express;
          a^.suivant:=stack^.suivant^.suivant^.suivant;
          stack:= a;
          dispose (b);
          effacer_arbre (c);
     END ELSE IF ((a^.nom2=variable) AND (c^.nom2=variable)
               AND (a^.puiss^.nomi=c^.puiss^.nomi)
               AND (a^.puiss^.denomi=c^.puiss^.denomi))
               OR
               ((a^.nom2=id_fonc) AND (c^.nom2=id_fonc)
               AND (a^.puiss^.nomi=c^.puiss^.nomi)
               AND (a^.puiss^.denomi=c^.puiss^.denomi)
               AND (a^.nomf = c^.nomf)
               AND (identique(a^.suite,c^.suite))) THEN
          BEGIN
               a^.nom1:=express;
               a^.suivant:=stack^.suivant^.suivant^.suivant;
               moins(c^.cofacteur,a^.cofacteur,a^.cofacteur);
               dispose (b);
               effacer_arbre (c);
```

```
            stack:= a;
        END ELSE
    BEGIN
        IF a^.nom2=rat THEN
        BEGIN
            b^.nom2:=cplus;
            a^.valeur^.nomi:=a^.valeur^.nomi * -1;
        END;
        IF (a^.nom2=variable) OR (a^.nom2=id_fonc) THEN
        BEGIN
            b^.nom2:=cplus;
            a^.cofacteur^.nomi:=a^.cofacteur^.nomi * -1;
        END;
        b^.un:=c;
        b^.deux:=a;
        b^.suivant:=stack^.suivant^.suivant^.suivant;
        b^.nom1:=express;
        a^.suivant:=NIL;
        c^.suivant:=NIL;
        stack:=b;
    END;
END;

PROCEDURE reduce_3;{<express>:=<terme>}
{*****************}
BEGIN
    stack^.nom1:=express;
END;

PROCEDURE reduce_4;{<terme>:=<terme><*><facteur>}
{*****************}
VAR a,b,c :arbre;
cas_general:boolean;
BEGIN
    cas_general:=false;
    a:= stack;
    b:= stack^.suivant;
    c:= stack^.suivant^.suivant;
    IF (a^.nom2=rat) THEN
    BEGIN
       IF (c^.nom2=rat) THEN
       BEGIN
            fois(c^.valeur,a^.valeur,a^.valeur);
            a^.nom1:=terme;
            a^.suivant:=stack^.suivant^.suivant^.suivant;
            stack:= a;
            dispose (b);
            effacer_arbre (c);
       END
       ELSE IF (c^.nom2=variable) OR (c^.nom2=id_fonc) THEN
            BEGIN
                 fois(c^.cofacteur,a^.valeur,c^.cofacteur);
                 c^.nom1:=terme;
                 c^.suivant:=stack^.suivant^.suivant^.suivant;
                 stack:= c;
                 effacer_arbre (a);
                 dispose (b);
            END ELSE cas_general:=true;
    END ELSE IF (a^.nom2=variable) THEN
    BEGIN
```

```
          IF (c^.nom2=variable) THEN
          BEGIN
                plus(c^.puiss,a^.puiss,a^.puiss);
                fois(c^.cofacteur,a^.cofacteur,a^.cofacteur);
               a^.suivant:=stack^.suivant^.suivant^.suivant;
               a^.nom1:=terme;
               a:=var_to_rat(a);
               stack:=a;
               dispose (b);
               effacer_arbre (c);
          END
          ELSE IF (c^.nom2=rat) THEN
               BEGIN
                     fois(a^.cofacteur,c^.valeur,a^.cofacteur);
                     a^.nom1:=terme;
                     a^.suivant:=stack^.suivant^.suivant^.suivant;
                     stack:= a;
                     effacer_arbre (c);
                     dispose (b);
               END ELSE cas_general:=true;
     END ELSE IF (a^.nom2=id_fonc) THEN
     BEGIN
          IF (c^.nom2=id_fonc) AND (a^.nomf=c^.nomf)
             AND identique(a^.suite,c^.suite) THEN
          BEGIN
                plus(c^.puiss,a^.puiss,a^.puiss);
                fois(c^.cofacteur,a^.cofacteur,a^.cofacteur);
               a:=var_to_rat(a);
               a^.suivant:=stack^.suivant^.suivant^.suivant;
               a^.nom1:=terme;
               stack:=a;
               dispose (b);
               effacer_arbre (c);
          END
          ELSE IF (c^.nom2=rat) THEN
               BEGIN
                     fois(a^.cofacteur,c^.valeur,a^.cofacteur);
                     a^.nom1:=terme;
                     a^.suivant:=stack^.suivant^.suivant^.suivant;
                     stack:= a;
                     effacer_arbre (c);
                     dispose (b);
               END ELSE cas_general:=true;
     END ELSE cas_general:=true;
     IF cas_general THEN
     BEGIN
          b^.un:=c;
          b^.deux:=a;
          b^.suivant:=stack^.suivant^.suivant^.suivant;
          b^.nom1:=terme;
          a^.suivant:=NIL;
          c^.suivant:=NIL;
          stack:=b;
    END;
END;

PROCEDURE reduce_5;{<terme>:=<terme></><facteur>}
{*****************}
VAR a,b,c :arbre;
inter:real;
```

```
cas_general:boolean;
BEGIN
     cas_general:=false;
     a:= stack;
     b:= stack^.suivant;
     c:= stack^.suivant^.suivant;
     IF (a^.nom2=rat) THEN
     BEGIN
        IF (c^.nom2=rat) THEN
        BEGIN
             divis(c^.valeur,a^.valeur,a^.valeur);
             a^.nom1:=terme;
             a^.suivant:=stack^.suivant^.suivant^.suivant;
             stack:= a;
             dispose (b);
             effacer_arbre (c);
        END ELSE IF (c^.nom2=variable) OR (c^.nom2=id_fonc) THEN
        BEGIN
             divis(c^.cofacteur,a^.valeur,c^.cofacteur);
             c^.nom1:=terme;
             c^.suivant:=stack^.suivant^.suivant^.suivant;
             stack:= c;
             effacer_arbre (a);
             dispose (b);
        END ELSE cas_general:=true;
     END ELSE IF (a^.nom2=variable) THEN
     BEGIN
          IF (c^.nom2=variable) THEN
          BEGIN
                moins(c^.puiss,a^.puiss,a^.puiss);
                divis(c^.cofacteur,a^.cofacteur,a^.cofacteur);
               a:=var_to_rat(a);
               a^.suivant:=stack^.suivant^.suivant^.suivant;
               a^.nom1:=terme;
               stack:=a;
               dispose (b);
               effacer_arbre (c);
          END ELSE IF (c^.nom2=rat) THEN
          BEGIN
                divis(c^.valeur,a^.cofacteur,a^.cofacteur);
                a^.puiss^.nomi:=a^.puiss^.nomi * -1;
                a^.nom1:=terme;
                a^.suivant:=stack^.suivant^.suivant^.suivant;
                stack:= a;
                effacer_arbre (c);
                dispose (b);
          END ELSE cas_general:=true;
     END ELSE IF (a^.nom2=id_fonc) THEN
     BEGIN
          IF (c^.nom2=id_fonc) AND (a^.nomf=c^.nomf)
              AND identique(a^.suite,c^.suite) THEN
          BEGIN
                moins(c^.puiss,a^.puiss,a^.puiss);
                divis(c^.cofacteur,a^.cofacteur,a^.cofacteur);
               a:=var_to_rat(a);
               a^.suivant:=stack^.suivant^.suivant^.suivant;
               a^.nom1:=terme;
               stack:=a;
               dispose (b);
               effacer_arbre (c);
```

```
          END
          ELSE IF (c^.nom2=rat) THEN
          BEGIN
                divis(c^.valeur,a^.cofacteur,a^.cofacteur);
                a^.puiss^.nomi:=a^.puiss^.nomi * -1;
                a^.nom1:=terme;
                a^.suivant:=stack^.suivant^.suivant^.suivant;
                stack:= a;
                effacer_arbre (c);
                dispose (b);
          END ELSE cas_general:=true;
     END ELSE cas_general:=true;
     IF cas_general THEN
     BEGIN
          IF (a^.nom2=variable) OR (a^.nom2=id_fonc) THEN
          BEGIN
               a^.puiss^.nomi:=a^.puiss^.nomi * -1;
               b^.nom2:=cfois;
          END;
          IF (a^.nom2=rat) THEN
          BEGIN
               b^.nom2:=cfois;
               IF a^.valeur^.denomi>0 THEN
               BEGIN
                    inter:=a^.valeur^.nomi;
                    a^.valeur^.nomi:=a^.valeur^.denomi;
                    a^.valeur^.denomi:=inter;
                    IF inter<0 THEN
                    BEGIN
                         a^.valeur^.nomi:=a^.valeur^.nomi*-1;
                         a^.valeur^.denomi:=a^.valeur^.denomi*-1;
                    END;
               END ELSE IF (a^.valeur^.denomi=-1) AND (a^.valeur^.nomi<>0) THEN
                   a^.valeur^.nomi:=-1/a^.valeur^.nomi;
          END;
          b^.un:=c;
          b^.deux:=a;
          b^.suivant:=stack^.suivant^.suivant^.suivant;
          b^.nom1:=terme;
          a^.suivant:=NIL;
          c^.suivant:=NIL;
          stack:=b;
    END;
END;

PROCEDURE reduce_6;{<terme>:=<facteur>}
{*****************}
BEGIN
     stack^.nom1:=terme;
END;

PROCEDURE reduce_7;{<facteur>:=<facteur><^><cvaleur>}
{*****************}
VAR a,b,c :arbre;
BEGIN
     a:= stack;
     b:= stack^.suivant;
     c:= stack^.suivant^.suivant;
     IF (a^.nom2=rat) AND (c^.nom2=rat) THEN
     BEGIN
```

```
          puissance(c^.valeur,a^.valeur,c^.valeur);
          c^.nom1:=facteur;
          c^.suivant:=stack^.suivant^.suivant^.suivant;
          stack:= c;
          dispose (b);
          effacer_arbre (a);
     END ELSE
     IF ((c^.nom2=variable) OR (c^.nom2=id_fonc)) AND (a^.nom2=rat) THEN
     BEGIN
          fois(c^.puiss,a^.valeur,c^.puiss);
          puissance(c^.cofacteur,a^.valeur,c^.cofacteur);
          c:=var_to_rat(c);
          c^.nom1:=facteur;
          c^.suivant:=stack^.suivant^.suivant^.suivant;
          stack:= c;
          dispose (b);
          effacer_arbre (a);
     END ELSE
     BEGIN
          b^.un:=c;
          b^.deux:=a;
          b^.suivant:=stack^.suivant^.suivant^.suivant;
          b^.nom1:=facteur;
          a^.suivant:=NIL;
          c^.suivant:=NIL;
          stack:=b;
    END;
END;

PROCEDURE reduce_8;{<facteur>:=<cvaleur>}
{*****************}
BEGIN
     stack^.nom1:=facteur;
END;

PROCEDURE reduce_9;{<cvaleur>:=<(><express><)>}
{*****************}
VAR a,b,c:arbre;
BEGIN
     a:=stack;
     b:=stack^.suivant;
     c:=stack^.suivant^.suivant;
     b^.suivant:=stack^.suivant^.suivant^.suivant;
     b^.nom1:=cvaleur;
     stack:=b;
     dispose (a);
     dispose (c);
END;

PROCEDURE reduce_10;          {<cvaleur>:=<-><(><express><)>}
{******************}
VAR a,b,c,d,e:arbre;
BEGIN
     a:=stack;
     b:=stack^.suivant;
     c:=stack^.suivant^.suivant;
     IF b^.nom2=rat THEN
     BEGIN
        b^.valeur^.nomi:= b^.valeur^.nomi * - 1;
        b^.suivant:=c^.suivant^.suivant;
```

```
        b^.nom1:=cvaleur;
        stack:=b;
        dispose (a);
        dispose (c^.suivant);
        dispose (c);
     END ELSE
     IF (b^.nom2=variable) OR (b^.nom2=id_fonc) THEN
     BEGIN
        b^.cofacteur^.nomi:=b^.cofacteur^.nomi * - 1;
        b^.suivant:=c^.suivant^.suivant;
        b^.nom1:=cvaleur;
        stack:=b;
        dispose (a);
        dispose (c^.suivant);
        dispose (c);
     END ELSE
     BEGIN
          new(d);
          new(e);
          d^.nom2:=cfois;
          d^.nom1:=cvaleur;
          e^.nom1:=rat;
          e^.nom2:=rat;
          e^.suivant:=NIL;
          b^.suivant:=NIL;
          new(e^.valeur);
          e^.valeur^.nomi:=-1;
          e^.valeur^.denomi:=1;
          d^.un:=e;
          d^.deux:=b;
          d^.suivant:=c^.suivant^.suivant;
          stack:=d;
          dispose (a);
          dispose (c^.suivant);
          dispose (c);
     END;
END;

PROCEDURE reduce_11;          {<cvaleur>:=<fonc>}
{******************}
BEGIN
     stack^.nom1:=cvaleur;
END;

PROCEDURE reduce_12;          {<cvaleur>:=<->fonc>}
{******************}
VAR a,b:arbre;
BEGIN
     a:=stack;
     b:=stack^.suivant;
     IF a^.nom2=rat THEN
     BEGIN
        a^.nom1:=cvaleur;
        a^.valeur^.nomi:= a^.valeur^.nomi * - 1;
        a^.suivant:=stack^.suivant^.suivant;
        stack:=a;
        dispose (b);
     END ELSE
     IF (a^.nom2=variable) OR (a^.nom2=id_fonc) THEN
     BEGIN
```

```
        a^.cofacteur^.nomi:=a^.cofacteur^.nomi * - 1;
        a^.suivant:=stack^.suivant^.suivant;
        a^.nom1:=cvaleur;
        stack:=a;
        dispose (b);
     END;
END;

PROCEDURE reduce_13;          {<cvaleur>:=<rat>}
{******************}
BEGIN
     stack^.nom1:=cvaleur;
END;

PROCEDURE reduce_14;          {<cvaleur>:=<->rat>}
{******************}
VAR a:arbre;
BEGIN
     a:=stack^.suivant;
     stack^.suivant:=stack^.suivant^.suivant;
     stack^.nom1:=cvaleur;
     stack^.valeur^.nomi:= stack^.valeur^.nomi * - 1;
     dispose (a);
END;

PROCEDURE reduce_15;          {<cvaleur>:=<variable>}
{******************}
BEGIN
     stack^.nom1:=cvaleur;
END;

PROCEDURE reduce_16;          {<cvaleur>:=<->variable>}
{******************}
VAR a,b:arbre;
BEGIN
     a:=stack;
     b:=stack^.suivant;
     IF a^.nom2=rat THEN
     BEGIN
        a^.nom1:=cvaleur;
        a^.valeur^.nomi:= a^.valeur^.nomi * - 1;
        a^.suivant:=stack^.suivant^.suivant;
        stack:=a;
        dispose (b);
     END ELSE
     IF (a^.nom2=variable) OR (a^.nom2=id_fonc) THEN
     BEGIN
        a^.cofacteur^.nomi:=a^.cofacteur^.nomi * - 1;
        a^.suivant:=stack^.suivant^.suivant;
        a^.nom1:=cvaleur;
        stack:=a;
        dispose (b);
     END;
END;

PROCEDURE reduce_17;          {<fonc>:=<id_fonc><cvaleur>}
{******************}
VAR a:real;
b:fonction;
inter:arbre;
```

```
BEGIN {------------------ debut de la procedure reduce_17 -----------}
     b:=stack^.suivant^.nomf;
     IF stack^.nom2=rat THEN
     BEGIN
          IF stack^.valeur^.denomi=-1 THEN a:=stack^.valeur^.nomi
          ELSE a:=stack^.valeur^.nomi/stack^.valeur^.denomi;
          new(nouveau);
          nouveau^.nom1:=rat;
          nouveau^.nom2:=rat;
          new(nouveau^.valeur);
          nouveau^.valeur^.denomi:=-1;
          IF b= FSIN  THEN nouveau^.valeur^.nomi:=SIN(a)
          ELSE IF b= FCOS   THEN nouveau^.valeur^.nomi:=COS(a)
          ELSE IF b= FTG    THEN nouveau^.valeur^.nomi:=tg(a)
          ELSE IF b= FCOTG  THEN nouveau^.valeur^.nomi:=cotg(a)
          ELSE IF b= FASIN  THEN nouveau^.valeur^.nomi:=asin(a)
          ELSE IF b= FACOS  THEN nouveau^.valeur^.nomi:=acos(a)
          ELSE IF b= FATG   THEN nouveau^.valeur^.nomi:=arctan(a)
          ELSE IF b= FACOTG THEN nouveau^.valeur^.nomi:=acotg(a)
          ELSE IF b= FLN    THEN nouveau^.valeur^.nomi:=ln(a)
          ELSE IF b= FEXP   THEN nouveau^.valeur^.nomi:=exp(a);
          IF (stack^.suivant^.cofacteur^.nomi<>1) THEN
             fois(nouveau^.valeur,stack^.suivant^.cofacteur,nouveau^.valeur);
          IF (stack^.suivant^.puiss^.nomi<>1) THEN
             puissance(nouveau^.valeur,stack^.suivant^.puiss,nouveau^.valeur);
          nouveau^.suivant:=stack^.suivant^.suivant;
          effacer_arbre (stack^.suivant);
          effacer_arbre (stack);
          stack:=nouveau;
     END ELSE
     BEGIN
          inter:=stack^.suivant;
          inter^.nom1:=fonc;
          inter^.suite:=stack;
          stack:=inter;
     END;
END; {------------------ fin de la procedure produce_17 -------------}
BEGIN {------------------ debut de la procedure reduce ---------------}
      CASE production OF
           0:BEGIN
                   erreur:=true;
                   stop_erreur;
                   exit;
             END;
           1:reduce_1;
           2:reduce_2;
           3:reduce_3;
           4:reduce_4;
           5:reduce_5;
           6:reduce_6;
           7:reduce_7;
           8:reduce_8;
           9:reduce_9;
           10:reduce_10;
           11:reduce_11;
           12:reduce_12;
           13:reduce_13;
           14:reduce_14;
```

```
          15:reduce_15;
          16:reduce_16;
          17:reduce_17;
      END;
END; {------------------- fin de la procedure reduce -----------------}


BEGIN {------------------ debut fonction do_arbre --------------------}
entree:=donnee^.message+'#';
donnee^.message:='ERROR';
fin_entree:=false;
erreur:=false;
strerror:='Erreur !';
index:=1;
tampon:=litchar;
new(stack);
stack^.nom1:=vide;
stack^.suivant:=NIL;
new(input);
input^.nom1:=vide;
input^.suivant:=NIL;
WHILE(stack^.nom1=vide)AND(NOT(fin_entree))AND (NOT erreur) DO shift;
IF erreur THEN
BEGIN
     stop_erreur;
     exit;
END;
IF fin_entree AND (stack^.nom1=vide)THEN
BEGIN
     erreur:=true;
     strerror:='Rien en entrée !';
     stop_erreur;
     exit;
END;
REPEAT
      BEGIN
           CASE sre[stack^.nom1,input^.nom1] OF
                r:BEGIN
                       reduce;
                  END;
                s:BEGIN
                       shift;
                  END;
                u:proc_s1;
                ELSE BEGIN
                          strerror:='Expression incorecte !';
                          stop_erreur;
                          exit;
                     END;
           END;
      END;

UNTIL (input^.nom1=vide) OR (erreur=true);
IF erreur THEN
BEGIN
     stop_erreur;
     exit;
END ELSE IF sre[stack^.nom1,vide]=e THEN
    BEGIN
         strerror:='Fin expression innattendue !';
```

```
          stop_erreur;
          exit;
     END;

WHILE (stackˆ.nom1<>express)DO
BEGIN
     reduce;
     IF erreur THEN
     BEGIN
          stop_erreur;
          exit;
     END;
END;
IF (stackˆ.suivantˆ.nom1<>vide) THEN
BEGIN
     IF stackˆ.suivantˆ.nom1=gpar THEN
        strerror:='Niveau parenthèses incorrect !'
     ELSE strerror:='Erreur d''expression !';
     erreur:=true;
     stop_erreur;
     exit;
END;
IF NOT erreur THEN
BEGIN
     donneeˆ.message:='ok';
     donneeˆ.boom:=stack;
END;
END; {------------------- fin fonction do_arbre ----------------------}


END.
```

```
PROGRAM derivation;
{*****************}
uses derivee1;

VAR ligne:STRING;
state:pointer;
prunier:arbre2;
fprim:arbre;

FUNCTION copy_arbre(poirier:arbre):arbre; { Cree, recopie et retourne un arbre}
{************************************} { semblable a celui passe en entree }
VAR aux:arbre;
    cvaleur,ccofacteur,cpuiss:prationnel;
BEGIN
   IF poirier^.nom2 IN [rat..cexp] THEN
   BEGIN
     CASE poirier^.nom2 OF
     rat:BEGIN
              new(aux);
              new(cvaleur);
              cvaleur^:=poirier^.valeur^;
              aux^.nom2:=rat;
              aux^.valeur:=cvaleur;
              aux^.suivant:=NIL;
              copy_arbre:=aux;
         END;
     variable:
         BEGIN
              new(aux);
              new(ccofacteur);
              new(cpuiss);
              ccofacteur^:=poirier^.cofacteur^;
              cpuiss^:=poirier^.puiss^;
              aux^.nom2:=variable;
              aux^.suivant:=NIL;
              aux^.cofacteur:=ccofacteur;
              aux^.puiss:=cpuiss;
              copy_arbre:=aux;
         END;
     cplus..cexp:
         BEGIN
              new(aux);
              aux^.suivant:=NIL;
              aux^.nom2:=poirier^.nom2;
              aux^.un:=copy_arbre(poirier^.un);
              aux^.deux:=copy_arbre(poirier^.deux);
              copy_arbre:=aux;
         END;
     id_fonc:
         BEGIN
              new(aux);
              aux^.suivant:=NIL;
              aux^.nom2:=id_fonc;
              aux^.nomf:=poirier^.nomf;
              new(ccofacteur);
              new(cpuiss);
              ccofacteur^:=poirier^.cofacteur^;
              cpuiss^:=poirier^.puiss^;
              aux^.cofacteur:=ccofacteur;
              aux^.puiss:=cpuiss;
```

```
                aux^.suite:=copy_arbre(poirier^.suite);
                copy_arbre:=aux;
            END;
        END;
    END ELSE copy_arbre:=NIL;
END;


FUNCTION opposer(entree:arbre):arbre;
{*********************************}
VAR aux:arbre;
BEGIN
     CASE entree^.nom2 OF
     rat:entree^.valeur^.nomi:=-1 * entree^.valeur^.nomi;
     variable,id_fonc:
          BEGIN
               entree^.cofacteur^.nomi:=-1 * entree^.cofacteur^.nomi;
          END;
     cdiv,cfois:
          BEGIN
               IF (entree^.un^.nom2=rat) THEN
               BEGIN
                    IF (entree^.un^.valeur^.nomi = -1)
                    AND (entree^.un^.valeur^.denomi = 1) THEN
                    BEGIN
                         aux:=entree^.deux;
                         effacer_arbre(entree^.un);
                         dispose(entree);
                         entree:=aux;
                    END;
               END ELSE IF (entree^.deux^.nom2=rat) THEN
               BEGIN
                    IF (entree^.deux^.valeur^.nomi = -1)
                    AND (entree^.deux^.valeur^.denomi = 1) THEN
                    BEGIN
                         aux:=entree^.un;
                         effacer_arbre(entree^.deux);
                         dispose(entree);
                         entree:=aux;
                    END;
               END ELSE entree^.un:=opposer(entree^.un);
          END;
     cplus,cmoins:
          BEGIN
               entree^.un:=opposer(entree^.un);
               entree^.deux:=opposer(entree^.deux);
          END;
     cexp:BEGIN
               new(aux);
               aux^.nom2:=cfois;
               aux^.suivant:=NIL;
               aux^.un:=entree;
               new(aux^.deux);
               aux^.deux^.nom2:=rat;
               aux^.deux^.suivant:=NIL;
               new(aux^.deux^.valeur);
               aux^.deux^.valeur^.nomi:=-1;
               aux^.deux^.valeur^.denomi:=1;
               entree:=aux;
          END;
```

```
     END;
     opposer:=entree;
END;

FUNCTION inverser(entree:arbre):arbre;
{**********************************}
VAR aux:arbre;
BEGIN
     CASE entree^.nom2 OF
     rat:inverser_rat(entree^.valeur);
     variable,id_fonc:
          BEGIN
               inverser_rat(entree^.cofacteur);
               entree^.puiss^.nomi:=-1*entree^.puiss^.nomi;
          END;
     cplus,cmoins:
          BEGIN
               new(aux);
               aux^.nom2:=cexp;
               aux^.suivant:=NIL;
               aux^.un:=entree;
               new(aux^.deux);
               aux^.deux^.nom2:=rat;
               new(aux^.deux^.valeur);
               aux^.deux^.valeur^.nomi:=-1;
               aux^.deux^.valeur^.denomi:=1;
               aux^.deux^.suivant:=NIL;
               entree:=aux;
          END;
     cfois:BEGIN
               entree^.un:=inverser(entree^.un);
               entree^.deux:=inverser(entree^.deux);
          END;
     cdiv:BEGIN
               aux:=entree^.un;
               entree^.un:=entree^.deux;
               entree^.deux:=aux;
          END;
     cexp:BEGIN
               IF (entree^.deux^.nom2=rat) THEN
               IF (entree^.deux^.valeur^.nomi = -1)
               AND (entree^.deux^.valeur^.denomi = 1) THEN
               BEGIN
                    entree:=entree^.un;
                    effacer_arbre(entree^.deux);
                    dispose(entree);
               END ELSE
               entree^.deux:=opposer(entree^.deux);
          END;
     END;
     inverser:=entree;
END;

FUNCTION rectifier(poirier:arbre;nom:arbre_symbol):arbre;
{*****************************************************}
VAR aux,aux2:arbre;
BEGIN
     CASE nom OF
     cplus,cfois:
          BEGIN
```

```
                IF (poirier^.deux^.nom2 = nom) THEN
                BEGIN
                     aux:=poirier^.deux;
                     WHILE (aux^.un^.nom2 = nom) DO aux:=aux^.un;
                     aux2:=aux^.un;
                     aux^.un:=poirier^.un;
                     poirier^.un:=poirier^.deux;
                     poirier^.deux:=aux2;
                END;
           END;
      cmoins:
           BEGIN
                WHILE (poirier^.deux^.nom2 = cplus) DO
                BEGIN
                     poirier^.deux^.deux:=opposer(poirier^.deux^.deux);
                     aux:=poirier^.deux;
                     poirier^.deux:=poirier^.deux^.un;
                     aux^.un:=poirier^.un;
                     poirier^.un:=aux;
                END;
                poirier^.deux:=opposer(poirier^.deux);
                poirier^.nom2:=cplus;
           END;
      cdiv:
           BEGIN
                WHILE (poirier^.deux^.nom2 = cfois) DO
                BEGIN
                     poirier^.deux^.deux:=inverser(poirier^.deux^.deux);
                     aux:=poirier^.deux;
                     poirier^.deux:=poirier^.deux^.un;
                     aux^.un:=poirier^.un;
                     poirier^.un:=aux;
                END;
                poirier^.deux:=inverser(poirier^.deux);
                poirier^.nom2:=cfois;
           END;
      END;
      rectifier:=poirier;
END;

FUNCTION trie_arbre(entree:arbre):arbre;
{************************************}
TYPE
liste_feuille=^type_liste_feuille;
type_liste_feuille=RECORD
                        elem:arbre;
                        next:liste_feuille;
                   END;


VAR
chaine,aux,prem,dern,dernier,atrier:liste_feuille;
a,b,c:arbre;
state2:pointer;

PROCEDURE effacer_chaine(entree:liste_feuille);
{*****************************************}
BEGIN
     IF (entree^.next<>NIL) THEN effacer_chaine(entree^.next);
     dispose(entree);
```

```
END;

FUNCTION do_chaine(poirier:arbre;nom:arbre_symbol):liste_feuille;
{************************************************************}
VAR aux,aux2:liste_feuille;

BEGIN
     new(aux);
     IF poirier^.nom2 = nom THEN
     BEGIN
          aux^.elem:=poirier^.deux;
          aux^.next:=do_chaine(poirier^.un,nom);
     END ELSE
     BEGIN
          aux^.elem:=poirier;
          aux^.next:=NIL;
     END;
     do_chaine:=aux;
END;

PROCEDURE ordonne(chaine:liste_feuille);
{************************************}
VAR aux:arbre;

FUNCTION intervertir(un,deux:arbre):boolean;
{******************************************}
VAR aux:arbre;
PROCEDURE echanger(un,deux:arbre);
{*******************************}
VAR aux:arbre;
BEGIN
     new(aux);
     aux^:=un^;
     un^:=deux^;
     deux^:=aux^;
     dispose(aux);
END;

BEGIN   {******************  debut fonction intervertir ***************}
     intervertir:=false;
     IF (un<>NIL) AND (deux<>NIL) THEN
     BEGIN IF ((un^.nom2 IN [rat..id_fonc]) OR (deux^.nom2 IN [rat..id_fonc]))
        THEN
          IF (un^.nom2 > deux^.nom2)
          OR ((un^.nom2 = deux^.nom2)
          AND
              (((un^.nom2 = id_fonc) AND
               (un^.nomf > deux^.nomf))
          OR ((un^.nom2 = id_fonc) AND
               (un^.nomf = deux^.nomf) AND
               (deux^.puiss^.nomi < un^.puiss^.nomi))
          OR ((un^.nom2 = variable) AND
               (deux^.puiss^.nomi < un^.puiss^.nomi))))

          THEN
          BEGIN
               echanger(un,deux);
               intervertir:=true;
          END
     END ELSE
```

```
        IF ((un^.nom2 IN [cplus..cexp]) AND (deux^.nom2 IN [cplus..cexp]))
        THEN IF NOT ((un^.nom2 = cfois) OR (a^.nom2 = cexp)
        AND (identique(un^.un,deux))) THEN
        BEGIN
             echanger(un,deux);
             intervertir:=true;
        END;
END;  {****************  fin fonction Intervertir  ******************}

BEGIN     {*************** debut procedure Ordonne **************}
     IF chaine^.next <> NIL THEN
     BEGIN
          prem:=chaine^.next;
          dern:=NIL;
          atrier:=chaine;
          WHILE (dern <> atrier) DO
          BEGIN
               atrier:=chaine;
               IF (atrier<>prem) THEN WHILE atrier^.next <> prem DO
                    atrier:=atrier^.next;
               prem:=NIL;
               IF atrier<>NIL THEN
               BEGIN
                    WHILE (atrier^.next<>NIL) AND (atrier <> dern) DO
                    BEGIN
                         IF intervertir(atrier^.elem,atrier^.next^.elem) THEN
                         BEGIN
                              IF prem=NIL THEN prem:=atrier;
                              dernier:=atrier;
                         END;
                         atrier:=atrier^.next;
                    END;
                    dern:=dernier;
                    atrier:=chaine;
                    dernier:=atrier;
               END ELSE
               BEGIN
                    atrier:=chaine;
                    dern:=atrier;
               END;
          END;
     END;
END;   {********************  Fin fonction Ordonne  *****************}

BEGIN  {******************  debut fonction trie_arbre  ***************}
     IF (entree^.nom2 = cplus) OR (entree^.nom2 = cfois) THEN
     BEGIN
          chaine:=do_chaine(entree,entree^.nom2);
          ordonne(chaine);
          effacer_chaine(chaine);
     END;
     trie_arbre:=entree;
END;


FUNCTION reduire_arbre(pommier:arbre):arbre;
{****************************************}
VAR a,b,c,aa,aux :arbre;
b_a_inverser:boolean;
CONST fonc_inv:ARRAY[FRIEN..FEXP]OF fonction =
```

```
      (FRIEN,FASIN,FACOS,FATG,FACOTG,FSIN,FCOS,FTG,FCOTG,FEXP,FLN);


FUNCTION elaguer(entree:arbre;nom:arbre_symbol):arbre;
{************************************************}
VAR a,b,c,aux,aux1,aux2:arbre;
fin:boolean;
rat_un,rat_a,rat_b:prationnel;

BEGIN
IF (nom=cplus) THEN
BEGIN
     new(rat_un);
     rat_un^.nomi:=1;
     rat_un^.denomi:=1;
     a:= entree^.un;
     b:= entree^.deux;
     c:=entree^.un;
     IF (a^.nom2=cplus) THEN
     BEGIN
          a:=a^.deux;
          fin:=false;
     END ELSE fin:=true;
     IF ((a^.nom2 IN[rat..id_fonc]) AND (b^.nom2 IN[rat..id_fonc])) THEN
     BEGIN
          IF (a^.nom2=b^.nom2) THEN
          BEGIN
               IF (a^.nom2=rat) THEN
               BEGIN
                    plus(a^.valeur,b^.valeur,a^.valeur);
                    effacer_arbre(b);
                    dispose(entree);
                    IF (a^.valeur^.nomi = 0) THEN
                    BEGIN
                         IF fin THEN entree:=a ELSE
                         BEGIN
                              effacer_arbre(a);
                              entree:=elaguer(c^.un,cplus);
                              dispose(c);
                         END;
                    END ELSE
                    BEGIN
                         IF fin THEN entree:=a
                         ELSE entree:=elaguer(c,cplus);
                    END;
               END ELSE IF ((a^.nom2=variable)
                   AND (a^.puiss^.nomi=b^.puiss^.nomi))
                   OR
                   ((a^.nom2=id_fonc)
                   AND (a^.puiss^.nomi=b^.puiss^.nomi)
                   AND (a^.nomf = b^.nomf)
                   AND (identique(a^.suite,b^.suite))) THEN
               BEGIN
                    plus(a^.cofacteur,b^.cofacteur,a^.cofacteur);
                    effacer_arbre(b);
                    dispose(entree);
                    IF (a^.cofacteur^.nomi = 0) THEN
                    BEGIN
                         IF fin THEN
                         BEGIN
```

```
                    effacer_arbre(a);
                    effacer_arbre(b);
                    new(aux);
                    aux^.nom2:=rat;
                    new(aux^.valeur);
                    aux^.valeur^.nomi:=0;
                    aux^.valeur^.denomi:=1;
                    aux^.suivant:=NIL;
                    entree:=aux;
               END ELSE
               BEGIN
                    effacer_arbre(a);
                    entree:=elaguer(c^.un,cplus);
                    dispose(c);
               END;
          END ELSE
          BEGIN
               IF fin THEN entree:=a
               ELSE entree:=elaguer(c,cplus);
          END;
     END;
END ELSE IF NOT fin THEN entree^.un:=elaguer(entree^.un,cplus)
          ELSE entree:=entree;

END ELSE IF ((a^.nom2 IN [cplus..cexp]) AND (b^.nom2 IN [cplus..cexp]))
    THEN
BEGIN
     IF identique(a,b) THEN
     BEGIN
          new(aux);
          aux^.nom2:=cfois;
          aux^.suivant:=NIL;
          aux^.un:=a;
          a:=aux;
          new(aux);
          aux^.nom2:=rat;
          aux^.suivant:=NIL;
          new(aux^.valeur);
          aux^.valeur^.nomi:=2;
          aux^.valeur^.denomi:=1;
          a^.deux:=aux;
          a:=reduire_arbre(a);
          effacer_arbre(b);
          dispose(entree);
          IF fin THEN entree:=a ELSE
          entree:=elaguer(c,cplus);
     END ELSE IF ((a^.nom2=cfois) AND (b^.nom2=cfois)
         AND (a^.deux^.nom2 IN [variable..id_fonc])
         AND (b^.deux^.nom2 IN [variable..id_fonc]))
         THEN
     BEGIN
          new(rat_a);
          rat_a^.nomi:=1;
          rat_a^.denomi:=1;
          new(rat_b);
          rat_b^.nomi:=1;
          rat_b^.denomi:=1;
          aux1:=a;
          WHILE ((aux1^.deux^.nom2 IN [variable..id_fonc])
                AND (aux1^.un^.nom2 = cfois)
```

```
                AND (aux1^.un^.deux^.nom2 IN[variable..id_fonc]))
          DO aux1:=aux1^.un;
          IF (aux1^.un^.nom2 IN [variable..id_fonc]) THEN
               aux1:=aux1^.un ELSE aux1:=aux1^.deux;
          fois(rat_a,aux1^.cofacteur,rat_a);
          aux1^.cofacteur^.nomi:=1;
          aux1^.cofacteur^.denomi:=1;
          aux2:=b;
          WHILE ((aux2^.deux^.nom2 IN [variable..id_fonc])
                AND (aux2^.un^.nom2 = cfois)
                AND (aux2^.un^.deux^.nom2 IN [variable..id_fonc]))
          DO aux2:=aux2^.un;
          IF (aux2^.un^.nom2 IN [variable..id_fonc]) THEN
              aux2:=aux2^.un ELSE aux2:=aux2^.deux;
          fois(rat_b,aux2^.cofacteur,rat_b);
          aux2^.cofacteur^.nomi:=1;
          aux2^.cofacteur^.denomi:=1;
          IF identique(a,b) THEN
          BEGIN
               plus(rat_a,rat_b,rat_a);
               new(aux);
               aux^.nom2:=cfois;
               aux^.suivant:=NIL;
               aux^.un:=a;
               new(aux^.deux);
               aux^.deux^.nom2:=rat;
               aux^.deux^.suivant:=NIL;
               aux^.deux^.valeur:=rat_a;
               dispose(rat_b);
               effacer_arbre(b);
               a:=reduire_arbre(aux);
               IF NOT fin THEN c^.deux:=a;
               dispose(entree);
               IF fin THEN entree:=a ELSE entree:=elaguer(c,cplus);
          END ELSE
          BEGIN
               fois(rat_a,aux1^.cofacteur,aux1^.cofacteur);
               fois(rat_b,aux2^.cofacteur,aux2^.cofacteur);
               dispose(rat_a);
               dispose(rat_b);
          END;
     END ELSE IF ((b^.nom2 = cfois) AND (identique(b^.un,a))) THEN
       BEGIN
          plus(rat_un,b^.deux^.valeur,b^.deux^.valeur);
          b:=reduire_arbre(b);
          IF fin THEN
          BEGIN
               effacer_arbre(a);
               dispose(entree);
               entree:=b;
          END ELSE
          BEGIN
               entree^.un:=c^.un;
               effacer_arbre(a);
               dispose(c);
               entree:=elaguer(entree,cplus);
          END;
       END ELSE IF NOT fin THEN entree^.un:=elaguer(entree^.un,cplus)
              ELSE entree:=entree;
END ELSE IF NOT fin THEN entree^.un:=elaguer(entree^.un,cplus)
```

```
            ELSE entree:=entree;
     elaguer:=entree;
END  {fin elaguer - cas PLUS - }

{*************************************************************************}
ELSE IF (nom=cfois) THEN
BEGIN
     new(rat_un);
     rat_unˆ.nomi:=1;
     rat_unˆ.denomi:=1;
     a:= entreeˆ.un;
     b:= entreeˆ.deux;
     c:=entreeˆ.un;
     IF (aˆ.nom2=cfois) THEN
     BEGIN
          a:=aˆ.deux;
          fin:=false;
     END ELSE fin:=true;
     IF ((aˆ.nom2 IN[rat..id_fonc]) AND (bˆ.nom2 IN[rat..id_fonc])) THEN
     BEGIN
          IF (bˆ.nom2 = rat) THEN
          BEGIN
               IF (aˆ.nom2 = rat) THEN
               BEGIN
                    fois(aˆ.valeur,bˆ.valeur,aˆ.valeur);
                    effacer_arbre(b);
                    dispose(entree);
                    IF (aˆ.valeurˆ.nomi = 0) THEN
                    BEGIN
                         entree:=a;
                         IF NOT fin THEN effacer_arbre(cˆ.un);
                    END ELSE IF (aˆ.valeurˆ.nomi = 1)
                              AND (aˆ.valeurˆ.denomi = 1) THEN
                    BEGIN
                         IF fin THEN entree:=a ELSE
                         BEGIN
                              effacer_arbre(a);
                              entree:=elaguer(cˆ.un,cfois);
                              dispose(c);
                         END;
                    END ELSE
                    BEGIN
                         IF fin THEN entree:=a
                         ELSE entree:=elaguer(c,cfois);
                    END;
               END ELSE {aˆ.nom2 = variable ou id_fonc}
               BEGIN
                    fois(aˆ.cofacteur,bˆ.valeur,aˆ.cofacteur);
                    effacer_arbre(b);
                    dispose(entree);
                    IF fin THEN entree:=a
                    ELSE entree:=elaguer(c,cfois);
               END;
          END ELSE { (bˆ.nom2=variable) or (bˆ.nom2=id_fonc) }
                   IF (aˆ.nom2=bˆ.nom2) THEN
          BEGIN
               IF (aˆ.nom2=variable)
               OR ((aˆ.nom2=id_fonc) AND (aˆ.nomf=bˆ.nomf)
                   AND identique(aˆ.suite,bˆ.suite)) THEN
               BEGIN
```

```
                    fois(a^.cofacteur,b^.cofacteur,a^.cofacteur);
                    plus(a^.puiss,b^.puiss,a^.puiss);
                    effacer_arbre(b);
                    dispose(entree);
                    IF (a^.puiss^.nomi=0) THEN
                    BEGIN
                         a:=var_to_rat(a);
                         IF (a^.valeur^.nomi = 0) THEN
                         BEGIN
                              entree:=a;
                              IF NOT fin THEN effacer_arbre(c^.un);
                         END ELSE IF (a^.valeur^.nomi = 1)
                                   AND (a^.valeur^.denomi = 1) THEN
                         BEGIN
                              IF fin THEN entree:=a ELSE
                              BEGIN
                                   effacer_arbre(a);
                                   entree:=elaguer(c^.un,cfois);
                                   dispose(c);
                              END;
                         END ELSE
                         BEGIN
                              IF fin THEN entree:=a
                              ELSE entree:=elaguer(c,cfois);
                         END;
                    END ELSE IF(a^.cofacteur^.nomi=0) THEN
                    BEGIN
                         effacer_arbre(entree);
                         new(aux);
                         aux^.nom2:=rat;
                         new(aux^.valeur);
                         aux^.valeur^.nomi:=0;
                         aux^.valeur^.denomi:=1;
                         aux^.suivant:=NIL;
                         entree:=aux;
                    END ELSE
                    BEGIN
                         IF fin THEN entree:=a
                         ELSE entree:=elaguer(c,cfois);
                    END;
               END ELSE
               BEGIN
                    fois(a^.cofacteur,b^.cofacteur,a^.cofacteur);
                    b^.cofacteur^.nomi:=1;
                    b^.cofacteur^.denomi:=1;
                    IF NOT fin THEN entree^.un:=elaguer(entree^.un,cfois);
               END;
          END ELSE
          BEGIN
               fois(a^.cofacteur,b^.cofacteur,a^.cofacteur);
               b^.cofacteur^.nomi:=1;
               b^.cofacteur^.denomi:=1;
               IF NOT fin THEN entree^.un:=elaguer(entree^.un,cfois);
          END;
     END ELSE IF ((a^.nom2 IN [cplus..cexp]) AND (b^.nom2 IN [cplus..cexp]))
     THEN
     BEGIN
          IF identique(a,b) THEN
          BEGIN
               new(aux);
```

```
                aux^.nom2:=cexp;
                aux^.suivant:=NIL;
                aux^.un:=a;
                a:=aux;
                new(aux);
                aux^.nom2:=rat;
                aux^.suivant:=NIL;
                new(aux^.valeur);
                aux^.valeur^.nomi:=2;
                aux^.valeur^.denomi:=1;
                a^.deux:=aux;
                effacer_arbre(b);
                dispose(entree);
                IF fin THEN entree:=a ELSE
                entree:=elaguer(c,cfois);

           END ELSE IF ((b^.nom2 = cexp) AND (identique(b^.un,a))) THEN
           BEGIN
                plus(rat_un,b^.deux^.valeur,b^.deux^.valeur);
                b:=reduire_arbre(b);
                IF fin THEN
                BEGIN
                     effacer_arbre(a);
                     dispose(entree);
                     entree:=b;
                END ELSE
                BEGIN
                     entree^.un:=c^.un;
                     effacer_arbre(a);
                     dispose(c);
                     entree:=elaguer(entree,cfois);
                END;
           END ELSE IF NOT fin THEN entree^.un:=elaguer(entree^.un,cfois);
      END ELSE IF NOT fin THEN entree^.un:=elaguer(entree^.un,cfois);
      elaguer:=entree;


END; {fin elaguer - cas FOIS -}
END;

{*****************************************************************************}


BEGIN    {******************  debut fonction REDUIRE  ***************}
     WHILE ((pommier^.nom2 IN [cfois..cexp]) AND (pommier^.deux^.nom2=rat)
     AND (pommier^.deux^.valeur^.nomi=1)
     AND (pommier^.deux^.valeur^.denomi=1)) DO
     BEGIN
          aux:=pommier^.un;
          effacer_arbre(pommier^.deux);
          dispose(pommier);
          pommier:=aux;
     END;
     WHILE ((pommier^.nom2 IN [cplus..cmoins]) AND (pommier^.deux^.nom2=rat)
     AND (pommier^.deux^.valeur^.nomi=0)
     AND (pommier^.deux^.valeur^.denomi=1)) DO
     BEGIN
          aux:=pommier^.un;
          effacer_arbre(pommier^.deux);
          dispose(pommier);
```

```
     pommier:=aux;
END;
CASE pommier^.nom2 OF
cplus..cdiv:
     BEGIN
          pommier^.deux:=reduire_arbre(pommier^.deux);
          pommier^.un:=reduire_arbre(pommier^.un);
          pommier:=rectifier(pommier,pommier^.nom2);
          pommier:=trie_arbre(pommier);
          IF (pommier^.nom2=cplus)THEN
             pommier:=elaguer(pommier,cplus)
          ELSE IF (pommier^.nom2=cfois) THEN
             pommier:=elaguer(pommier,cfois);
     END;
cexp:
     BEGIN
          pommier^.deux:=reduire_arbre(pommier^.deux);
          pommier^.un:=reduire_arbre(pommier^.un);
          IF(pommier^.deux^.nom2=rat) THEN
          IF(pommier^.deux^.valeur^.nomi=0) THEN
          BEGIN
               new(aux);
               aux^.nom2:=rat;
               aux^.suivant:=NIL;
               new(aux^.valeur);
               aux^.valeur^.nomi:=1;
               aux^.valeur^.denomi:=1;
               effacer_arbre(pommier);
               pommier:=aux;
          END ELSE IF(pommier^.deux^.valeur^.nomi=1)
                    AND (pommier^.deux^.valeur^.denomi=1) THEN
          BEGIN
               aux:=pommier^.un;
               pommier^.un:=NIL;
               effacer_arbre(pommier^.deux);
               dispose(pommier);
               pommier:=aux;
          END ELSE IF(pommier^.deux^.valeur^.nomi=-1)
                    AND (pommier^.deux^.valeur^.nomi=1) THEN
          BEGIN
               pommier^.nom2:=cdiv;
               aux:=pommier^.un;
               pommier^.un:=pommier^.deux;
               pommier^.un^.valeur^.nomi:=1;
               pommier^.deux:=aux;
               pommier:=reduire_arbre(pommier);
          END ELSE IF (pommier^.un^.nom2 IN[variable..id_fonc]) THEN
          BEGIN
               fois(pommier^.un^.puiss,pommier^.deux^.valeur,
                    pommier^.un^.puiss);
               effacer_arbre(pommier^.deux);
               aux:=pommier^.un;
               dispose(pommier);
               pommier:=aux;
          END;
          IF(pommier^.un^.nom2=rat) THEN
          IF((pommier^.un^.valeur^.nomi=1)
          AND (pommier^.un^.valeur^.denomi=1))
          OR(pommier^.un^.valeur^.nomi=0)THEN
          BEGIN
```

```
                aux:=pommier^.un;
                pommier^.un:=NIL;
                effacer_arbre(pommier);
                pommier:=aux;
            END;
        END;
id_fonc:
        BEGIN
            pommier^.suite:=reduire_arbre(pommier^.suite);
            IF (pommier^.suite^.nom2=id_fonc) THEN
            IF (fonc_inv[pommier^.suite^.nomf]=pommier^.nomf)
            AND (pommier^.suite^.puiss^.nomi=1)
            AND (pommier^.suite^.puiss^.denomi=1)
            AND (pommier^.suite^.cofacteur^.nomi=1)
            AND (pommier^.suite^.cofacteur^.denomi=1) THEN
            BEGIN
                aux:=pommier^.suite^.suite;
                pommier^.suite^.suite:=NIL;
                new(a);
                a^.nom2:=rat;
                a^.suivant:=NIL;
                a^.valeur:=pommier^.puiss;
                new(b);
                b^.nom2:=cexp;
                b^.un:=aux;
                b^.deux:=a;
                new(aux);
                aux^.nom2:=cfois;
                aux^.suivant:=NIL;
                aux^.un:=b;
                new(a);
                a^.nom2:=rat;
                a^.suivant:=NIL;
                a^.valeur:=pommier^.cofacteur;
                aux^.deux:=a;
                dispose(pommier^.suite^.cofacteur);
                dispose(pommier^.suite^.puiss);
                dispose(pommier^.suite);
                dispose(pommier);
                pommier:=aux;
                pommier:=reduire_arbre(pommier);
            END;
            IF (pommier^.nom2=id_fonc) THEN
            IF ((pommier^.nomf=FEXP)
            AND (pommier^.suite^.nom2=id_fonc)
            AND (pommier^.suite^.nomf=FLN)
            AND (pommier^.suite^.puiss^.nomi=1)) THEN
            BEGIN
                new(aux);
                aux^.nom2:=cexp;
                aux^.suite:=NIL;
                aux^.un:=pommier^.suite^.suite;
                new(aux^.deux);
                aux^.deux^.nom2:=rat;
                aux^.deux^.suite:=NIL;
                aux^.deux^.valeur:=pommier^.suite^.cofacteur;
                fois(aux^.deux^.valeur,pommier^.puiss,aux^.deux^.valeur);
                new(a);
                a^.nom2:=cfois;
                a^.suite:=NIL;
```

```
                a^.un:=aux;
                new(b);
                b^.nom2:=rat;
                b^.suite:=NIL;
                a^.deux:=b;
                b^.valeur:=pommier^.cofacteur;
                dispose(pommier^.suite^.puiss);
                dispose(pommier^.suite);
                dispose(pommier);
                pommier:=reduire_arbre(a);
           END;
           IF (pommier^.nom2=id_fonc) AND (pommier^.nomf=FTG) THEN
           BEGIN
                new(aux);
                aux^.nom2:=cdiv;
                aux^.un:=pommier;
                aux^.un^.nomf:=FSIN;
                aux^.deux:=copy_arbre(pommier);
                aux^.deux^.nomf:=FCOS;
                pommier:=reduire_arbre(aux);
           END ELSE IF (pommier^.nom2=id_fonc) AND (pommier^.nomf=FCOTG) the
           BEGIN
                new(aux);
                aux^.nom2:=cdiv;
                aux^.un:=pommier;
                aux^.un^.nomf:=FCOS;
                aux^.deux:=copy_arbre(pommier);
                aux^.deux^.nomf:=FSIN;
                pommier:=reduire_arbre(aux);
           END;
     END;
variable:
     BEGIN
          IF (pommier^.puiss^.nomi=0) THEN
             pommier:=var_to_rat(pommier);
     END;
END;
IF (pommier^.nom2 IN [variable..id_fonc]) AND (pommier^.puiss^.nomi=0)
THEN
BEGIN
     new(aux);
     aux^.nom2:=rat;
     aux^.suivant:=NIL;
     new(aux^.valeur);
     aux^.valeur^:=pommier^.cofacteur^;
     effacer_arbre(pommier);
     pommier:=aux;
END;
IF (pommier^.nom2 IN [variable..id_fonc]) AND (pommier^.cofacteur^.nomi=0)
THEN
BEGIN
     new(aux);
     aux^.nom2:=rat;
     aux^.suivant:=NIL;
     new(aux^.valeur);
     aux^.valeur^.nomi:=0;
     aux^.valeur^.denomi:=1;
     effacer_arbre(pommier);
     pommier:=aux;
END;
```

```
     reduire_arbre:=pommier;

END;   {****************** fin fonction REDUIRE_ARBRE ***************}
{*****************************  DERIVATION  *************************}

FUNCTION derivee(poirier:arbre):arbre;
{***********************************}
VAR aux,aux2,aux3,sortie:arbre;
    rat_aux:prationnel;
BEGIN
     derivee:=NIL;
     CASE poirier^.nom2 OF
     rat:BEGIN
            new(aux);
            aux^.nom2:=rat;
            aux^.suivant:=NIL;
            new(aux^.valeur);
            aux^.valeur^.nomi:=0;
            aux^.valeur^.denomi:=1;
            sortie:=aux;
         END;
     variable:
         BEGIN
            new(aux);
            aux^.nom2:=variable;
            aux^.suivant:=NIL;
            new(aux^.cofacteur);
            new(aux^.puiss);
            fois(poirier^.cofacteur,poirier^.puiss,aux^.cofacteur);
            new(rat_aux);
            rat_aux^.nomi:=1;
            rat_aux^.denomi:=1;
            moins(poirier^.puiss,rat_aux,aux^.puiss);
            IF (aux^.puiss^.nomi=0) THEN aux:=var_to_rat(aux);
            sortie:=aux;
         END;
     cplus:
         BEGIN
            new(aux);
            aux^.nom2:=cplus;
            aux^.suivant:=NIL;
            aux^.un:=derivee(poirier^.un);
            aux^.deux:=derivee(poirier^.deux);
            sortie:=aux;
         END;
     cmoins:
         BEGIN
            new(aux);
            aux^.nom2:=cmoins;
            aux^.suivant:=NIL;
            aux^.un:=derivee(poirier^.un);
            aux^.deux:=derivee(poirier^.deux);
            sortie:=aux;
         END;
     cfois:
         BEGIN
            new(aux);
            aux^.nom2:=cplus;
            aux^.suivant:=NIL;
            new(aux2);
```

```
          aux2^.nom2:=cfois;
          aux2^.suivant:=NIL;
          aux^.un:=aux2;
          new(aux3);
          aux3^.nom2:=cfois;
          aux3^.suivant:=NIL;
          aux^.deux:=aux3;
          aux2^.un:=derivee(poirier^.un);
          aux2^.deux:=copy_arbre(poirier^.deux);
          aux3^.un:=derivee(poirier^.deux);
          aux3^.deux:=copy_arbre(poirier^.un);
          sortie:=aux;
     END;
cdiv:
     BEGIN
          new(aux);
          aux^.nom2:=cmoins;
          aux^.suivant:=NIL;
          new(aux2);
          aux2^.nom2:=cfois;
          aux2^.suivant:=NIL;
          aux^.un:=aux2;
          new(aux3);
          aux3^.nom2:=cfois;
          aux3^.suivant:=NIL;
          aux^.deux:=aux3;
          aux2^.un:=derivee(poirier^.un);
          aux2^.deux:=copy_arbre(poirier^.deux);
          aux3^.un:=derivee(poirier^.deux);
          aux3^.deux:=copy_arbre(poirier^.un);
          new(aux2);
          aux2^.nom2:=cdiv;
          aux2^.suivant:=NIL;
          aux2^.un:=aux;
          new(aux3);
          aux3^.nom2:=cexp;
          aux3^.suivant:=NIL;
          aux3^.un:=copy_arbre(poirier^.deux);
          new(aux);
          aux^.nom2:=rat;
          aux^.suivant:=NIL;
          new(aux^.valeur);
          aux^.valeur^.nomi:=2;
          aux^.valeur^.denomi:=1;
          aux3^.deux:=aux;
          sortie:=aux2;
     END;
cexp:
     BEGIN
          new(aux);
          aux^.suivant:=NIL;
          aux^.nom2:=cfois;
          aux^.un:=copy_arbre(poirier);
          new(aux2);
          aux2^.nom2:=cfois;
          aux2^.suivant:=NIL;
          aux2^.un:=copy_arbre(poirier^.deux);
          new(aux3);
          aux2^.deux:=aux3;
          aux3^.nom2:=id_fonc;
```

```
          aux3^.nomf:=FLN;
          aux3^.suivant:=NIL;
          new(rat_aux);
          rat_aux^.nomi:=1;
          rat_aux^.denomi:=1;
          aux3^.cofacteur:=rat_aux;
          new(rat_aux);
          rat_aux^.nomi:=1;
          rat_aux^.denomi:=1;
          aux3^.puiss:=rat_aux;
          aux3^.suite:=copy_arbre(poirier^.un);
          aux^.deux:=derivee(aux2);
          effacer_arbre(aux2);
          sortie:=aux;
     END;
id_fonc:
     BEGIN
          IF ((poirier^.cofacteur^.nomi<> 1)
          OR  (poirier^.cofacteur^.denomi <> 1)
          OR  (poirier^.puiss^.nomi<> 1)
          OR  (poirier^.puiss^.denomi <> 1)) THEN
          BEGIN
               new(aux);
               aux^.nom2:=cfois;
               aux^.suivant:=NIL;
               aux2:=copy_arbre(poirier);
               aux^.un:=aux2;
               fois(aux2^.cofacteur,aux2^.puiss,aux2^.cofacteur);
               new(rat_aux);
               rat_aux^.nomi:=1;
               rat_aux^.denomi:=1;
               moins(aux2^.puiss,rat_aux,aux2^.puiss);
               dispose(rat_aux);
               aux3:=copy_arbre(poirier);
               aux3^.cofacteur^.nomi:=1;
               aux3^.cofacteur^.denomi:=1;
               aux3^.puiss^.nomi:=1;
               aux3^.puiss^.denomi:=1;
               aux^.deux:=derivee(aux3);
               effacer_arbre(aux3);
               sortie:=aux;
          END ELSE  { cofacteur et exposant = 1 }
          BEGIN
               CASE poirier^.nomf OF
               FSIN:
                 BEGIN
                      new(aux);
                      aux^.nom2:=cfois;
                      aux^.suivant:=NIL;
                      aux^.un:=copy_arbre(poirier);
                      aux^.un^.nomf:=FCOS;
                      aux^.deux:=derivee(poirier^.suite);
                      sortie:=aux;
                 END;
               FCOS:
                 BEGIN
                      new(aux);
                      aux^.nom2:=cfois;
                      aux^.suivant:=NIL;
                      aux^.un:=copy_arbre(poirier);
```

```
        aux^.un^.nomf:=FSIN;
        aux^.un^.cofacteur^.nomi:=-1*aux^.un^.cofacteur^.nomi;
        aux^.deux:=derivee(poirier^.suite);
        sortie:=aux;
  END;
FTG:
  BEGIN
  END;
FCOTG:
  BEGIN
  END;
FASIN:
  BEGIN
        new(aux);
        aux^.nom2:=cexp;
        aux^.suivant:=NIL;
        new(aux2);
        aux2^.nom2:=rat;
        aux2^.suivant:=NIL;
        new(rat_aux);
        rat_aux^.nomi:=2;
        rat_aux^.denomi:=1;
        aux2^.valeur:=rat_aux;
        aux^.un:=copy_arbre(poirier^.suite);
        aux^.deux:=aux2;
        new(aux2);
        aux2^.nom2:=cmoins;
        aux2^.suivant:=NIL;
        new(aux3);
        aux3^.nom2:=rat;
        aux3^.suivant:=NIL;
        new(rat_aux);
        rat_aux^.nomi:=1;
        rat_aux^.denomi:=1;
        aux3^.valeur:=rat_aux;
        aux2^.un:=aux3;
        aux2^.deux:=aux;
        new(aux);
        aux^.nom2:=cexp;
        aux^.suivant:=NIL;
        aux^.un:=aux2;
        new(aux2);
        aux2^.suivant:=NIL;
        aux2^.nom2:=rat;
        new(rat_aux);
        rat_aux^.nomi:=-1;
        rat_aux^.denomi:=2;
        aux2^.valeur:=rat_aux;
        aux^.deux:=aux2;
        new(aux2);
        aux2^.nom2:=cfois;
        aux2^.suivant:=NIL;
        aux2^.un:=aux;
        aux2^.deux:=derivee(poirier^.suite);
        sortie:=aux2;
  END;
FACOS:
  BEGIN
        new(aux);
        aux^.nom2:=cfois;
```

```
        aux^.suivant:=NIL;
        aux2:=copy_arbre(poirier);
        aux2^.nomf:=FASIN;
        aux^.un:=derivee(aux2);
        effacer_arbre(aux2);
        new(aux3);
        aux3^.nom2:=rat;
        aux3^.suivant:=NIL;
        new(rat_aux);
        rat_aux^.nomi:=-1;
        rat_aux^.denomi:=1;
        aux3^.valeur:=rat_aux;
        aux^.deux:=aux3;
        sortie:=aux;
  END;
FATG:
  BEGIN
        new(aux);
        aux^.nom2:=cexp;
        aux^.suivant:=NIL;
        new(aux2);
        aux2^.nom2:=rat;
        aux2^.suivant:=NIL;
        new(rat_aux);
        rat_aux^.nomi:=2;
        rat_aux^.denomi:=1;
        aux2^.valeur:=rat_aux;
        aux^.un:=copy_arbre(poirier^.suite);
        aux^.deux:=aux2;
        new(aux2);
        aux2^.nom2:=cmoins;
        aux2^.suivant:=NIL;
        new(aux3);
        aux3^.nom2:=rat;
        aux3^.suivant:=NIL;
        new(rat_aux);
        rat_aux^.nomi:=1;
        rat_aux^.denomi:=1;
        aux3^.valeur:=rat_aux;
        aux2^.un:=aux3;
        aux2^.deux:=aux;
        new(aux);
        aux^.nom2:=cexp;
        aux^.suivant:=NIL;
        aux^.un:=aux2;
        new(aux2);
        aux2^.suivant:=NIL;
        aux2^.nom2:=rat;
        new(rat_aux);
        rat_aux^.nomi:=-1;
        rat_aux^.denomi:=1;
        aux2^.valeur:=rat_aux;
        aux^.deux:=aux2;
        new(aux2);
        aux2^.nom2:=cfois;
        aux2^.suivant:=NIL;
        aux2^.un:=aux;
        aux2^.deux:=derivee(poirier^.suite);
        sortie:=aux2;
  END;
```

```
                    FACOTG:
                      BEGIN
                           new(aux);
                           aux^.nom2:=cfois;
                           aux^.suivant:=NIL;
                           aux2:=copy_arbre(poirier);
                           aux2^.nomf:=FATG;
                           aux^.un:=derivee(aux2);
                           effacer_arbre(aux2);
                           new(aux3);
                           aux3^.nom2:=rat;
                           aux3^.suivant:=NIL;
                           new(rat_aux);
                           rat_aux^.nomi:=-1;
                           rat_aux^.denomi:=1;
                           aux3^.valeur:=rat_aux;
                           aux^.deux:=aux3;
                           sortie:=aux;
                      END;
                    FLN:
                      BEGIN
                           new(aux);
                           aux^.nom2:=cdiv;
                           aux^.suivant:=NIL;
                           aux^.un:=derivee(poirier^.suite);
                           aux^.deux:=copy_arbre(poirier^.suite);
                           sortie:=aux;
                      END;
                    FEXP:
                      BEGIN
                           new(aux);
                           aux^.nom2:=cfois;
                           aux^.suivant:=NIL;
                           aux^.un:=copy_arbre(poirier);
                           aux^.deux:=derivee(poirier^.suite);
                           sortie:=aux;
                      END;
                  END;

           END;
      END;
      derivee:=reduire_arbre(sortie);
END;

{***************************** MAIN PROGRAM ************************}
BEGIN
      ligne:='';
      new(pun);
      pun^:=1;
      writeln(' Entrer 000 pour terminer  !');
      WHILE ligne<>'000' DO
      BEGIN
           write (' Expression      = ');
           readln (ligne);
           IF ligne<>'000' THEN
           BEGIN
                mark(state);
                new(prunier);
                new(prunier^.boom);
```

```
                prunier^.message:=ligne;
                prunier^.boom:=NIL;
                do_arbre(prunier);
                IF prunier^.message='ok' THEN
                BEGIN
                     prunier^.boom:=reduire_arbre(prunier^.boom);
                     writeln(' Simplification  = ',lire_arbre(prunier^.boom,pun))
                     fprim:=derivee(prunier^.boom);
                     fprim:=reduire_arbre(fprim);
                     writeln(' Derivee         = ',lire_arbre(fprim,pun));
writeln;
effacer_arbre(prunier^.boom);
dispose(prunier);
                END ELSE writeln(prunier^.message);
                dispose(prunier);
                release(state);
          END;
     END;
END.
```